CSE2002 PROGRAMMING II
2020 – 2021 SUMMER

Laboratory Date : 03 August 2021

Topic : Advanced Uses of Pointers

Program : Inventory2

Definition : The program is a modification of the parts database program of Laboratory 27-07-2021 (the original program), with the database stored in a linked list this time.

Advantages of using a linked list:

- No need to put a limit on the size of the database.
- Database can easily be kept sorted by part number. In the original program, the database wasn't sorted.

The part structure will contain an additional member (a pointer to the next node) and *inventory* will point to the first node in the list.

Most of the functions in the new program will closely resemble their counterparts in the original program. *find_part* and *insert* will be more complex, however, since we'll keep the nodes in the *inventory* list sorted by part number.

In the original program, *find_part* returns an index into the *inventory* array. In the new program, *find_part* will return a pointer to the node that contains the desired part number. If it doesn't find the part number, *find_part* will return a null pointer. Since the list of parts is sorted, *find_part* can stop when it finds a node containing a part number that's greater than or equal to the desired part number. When the loop terminates, we'll need to test whether the part was found.

The original version of *insert* stores a new part in the next available array element. The new version must determine where the new part belongs in the list and insert it there. It will also check whether the part number is already present in the list.

Try to write *erase* function: The codes e (erase) will be used to represent delete operation. Prompts the user to enter a part number. Prints an error message if the part doesn't exist; otherwise, removes the part from the database.

CSE2002 PROGRAMMING II
2020 – 2021 SUMMER

Laboratory Date : 6 July 2021

Topic : Basics Types, Formatted I/O, Expressions, Selection Statements, Loops, Functions, Program Organization

---

Program : Guessing a Number

Definition : The program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible.

This program has several different tasks:

- ✓ **Initializing the random number generator:** Initializes the random number generator using the time of day. It relies on the *time()*, *srand()*, and *rand()* functions to generate random number.
- ✓ **Choosing a secret number:** Randomly selects a number between 1 and 100 and stores it in an external variable.
- ✓ **Interacting with the user:** Repeatedly reads user guesses and tells the user whether each is too low, too high, or correct. When the guess is correct, prints the total number of guesses and returns. The interaction will continue until the correct number is picked.

Expected output:

```
Guess the secret number between 1 and 100.
A new number has been chosen.
Enter guess: 50
Too high; try again.
Enter guess: 25
Too low; try again.
Enter guess: 40
Too high; try again.
Enter guess: 30
Too low; try again.
Enter guess: 35
Too low; try again.
Enter guess: 38
Too high; try again.
Enter guess: 36
Too low; try again.
Enter guess: 37
You won in 8 quesses!

Play again? (Y/N)y

A new number has been chosen.
Enter guess: 65
Too low; try again.
Enter guess: 85
Too low; try again.
Enter guess: 95
Too high; try again.
Enter guess: 90
Too low; try again.
Enter guess: 93
Too high; try again.
Enter guess: 91
You won in 6 quesses!

Play again? (Y/N)n
```

Modify the program:

This program uses an external variable *secret_number*. When we move *secret_number* as local variable into main function, you should alter **Choosing a secret number** task that returns the new number. You should rewrite **Interacting with the user** task that *secret_number* can be passed to it as an argument.

Topic : Pointers

Program : Selection Sort

Definition : The program sorts the elements of array in either ascending or descending order by using selection sort technique.

This program has several different tasks:

- ✓ **Sorting elements by using selection sort:** Selection sort algorithm divides the list into two parts:
  - the sorted part at the left end that is empty.
  - the unsorted part at the right end that is the entire list.

  In main program, the user selects sorting order either ascending or descending. According to sorting order, (1) the smallest (minimum) element is selected for ascending order (2) the biggest (maximum) element is selected for descending order, from unsorted array and swapped with the leftmost element. That element becomes a part of the sorted array. In every iteration of algorithm, either the minimum element (considering ascending order) or the maximum element (considering descending order) from the unsorted subarray is moved to the sorted subarray.

  Assume that the following array contains integer elements where LENGTH is the size of array.
  int my_array[LENGTH] = {12, 36, 27, 8, 77, 43, 22, 58, 60, 85};
  Sort the given array by using "**void selection_sort(int array[], char s_order)**" function.
- ✓ **Swapping elements:** Swap the first element with the second element by using "**void swap(int *first, int *second)**" function.
- ✓ **Printing array:** Print the given array by using "**void display_array(int array[])**" function.

Expected output:

Ascending order:

```
The array before sort:
12      36      27      8       77      43      22      58      60      85
Choose the sorting order (either ascending or descending) (A/D) : A
The array after selection sort:
8       12      22      27      36      43      58      60      77      85
```

Descending order:

```
The array before sort:
12      36      27      8       77      43      22      58      60      85
Choose the sorting order (either ascending or descending) (A/D) : D
The array after selection sort:
85      77      60      58      43      36      27      22      12      8
```

Topic : Pointers and Arrays

Program : Find mode, arithmetic mean, and harmonic mean

Definition : The program finds the mode of array and calculates both arithmetic and harmonic mean.

Mode: The most common value in a set of numerical values.

Median: The middle value if all values in a set of numerical values were ordered.

Arithmetic mean: The average of a set of numerical values, as calculated by adding them together and dividing by the number of terms in the set.

$$Arithmetic\ mean = \frac{\sum_{i=0}^{N} X_i}{N} \quad where\ N\ is\ the\ number\ of\ values\ in\ a\ set$$

Harmonic mean: The reciprocal of the arithmetic mean of a set of numerical values, as calculated the number of values divided by adding reciprocal of the values (one over each value).

$$Harmonic\ mean = \frac{N}{\sum_{i=0}^{N} \frac{1}{X_i}} \quad where\ N\ is\ the\ number\ of\ values\ in\ a\ set$$

Use the following functions prototype to develop main function.

int find_mode(int array[], int size);

void calculate_arithmetic_mean(int array[], int size, float *a_mean);

void calculate_harmonic_mean(int array[], int size, float *h_mean);

void printArray(int array[], int size)

Expected output:

```
***One-dimensional array***
32      32      32      32      34      32      26      28      28      29      31      24      27      25      27      28      30      29      29      29      31      32      31      29      31      31      35      33
The mode of array is 32
The arithmetic mean of array is 30.000000
The harmonic mean of array is 29.771446

***Multidimensional array***
4       4       82      34      56
5       34      76      90      76
2       6       1       2       45
The mode of the first row of the 2D array is 4
The arithmetic mean of the first row of the 2D array is 36.000000
The harmonic mean of the first row of the 2D array is 8.937125
```

Modify the program: Try to find the median of the array.

CSE2002 PROGRAMMING II
2020 – 2021 SUMMER

Laboratory Date : 13 July 2021

Topic : Strings

Program : Palindrome

Definition :

Palindrome: The letters in the message are the same from left to right as from right to left.

First program: The program reads a message and checks whether it is a palindrome.

Second program: The program checks command-line argument whether it is a palindrome.

Expected output:

```
Enter a message: He lived as a Devil, eh?
Palindrome
```

```
Enter a message: How are you?
Not a palindrome
```

```
Enter a message: madam
Palindrome
```

Topic : The Preprocessor

Program : Preprocessor

Definition : The program includes different macros such as simple, parameterized, predefined.

- Use predefined macros to print file information. “File information is given below” message is simple macro definition.
- Use parameterized macros
  - To add two numbers.
  - To subtract two numbers that are constant values.
  - To print “Hello John Doe. Welcome to CSE2002 Programming II Lab!”, define “message_for(name, surname)” by using # operator.

- Use Conditional Compilation to assign bigger number as first number before using subtraction macro.

Expected output:

```
File information is given below.
        File :preprocessor.c
        Date :Feb  5 2021
        Time :17:46:54
        Line :24
        ANSI :1
Hello John Doe. Welcome to CSE2002 Programming II Lab!
Enter first number:5
Enter second number:6
The summation of 5 and 6 is 11
The subtraction of 8 and 9 is calculated.
 9 - 8 = 1
```

Question: Show that the following program will look like after preprocessing.

```
#define N = 10
#define INC(x) x+1
#define SUB (x,y) x-y
#define SQR(x) ((x)*(x))
#define CUBE(x) (SQR(x)*(x))
#define M1(x,y) x##y
#define M2(x,y) #x #y

int main(void)
{
  int a[N], i, j, k, m;

#ifdef N
  i = j;
#else
  j = i;
#endif

  i = 10 * INC(j);

  i = SUB(j, k);
  i = SQR(SQR(j));
  i = CUBE(j);
  i = M1(j, k);
  puts(M2(i, j));

#undef SQR
  i = SQR(j);
#define SQR
  i = SQR(j);

  return 0;
}
```

CSE2002 PROGRAMMING II
2020 – 2021 SUMMER

Laboratory Date : 27 July 2021

Topic : Structures, Unions, and Enumerations

Program : Inventory

Definition : The program tracks parts stored in a warehouse. Information about the parts is stored in an array of structures. Contents of each structure: Part number, Name, Quantity.

Operations supported by the program:

- Add a new part number, part name, and initial quantity on hand. Prints an error message and returns prematurely if the part already exists or the database is full.
- Given a part number, print the name of the part and the current quantity on hand if the part number is found; otherwise prints an error message.
- Given a part number, change the quantity on hand if the part already exists; otherwise print an error message.
- Print a table showing all information in the database.
- Terminate program execution.

The program prompts the user to enter an operation code, then calls a function to perform the requested action. The codes i (insert), s (search), u (update), p (print), and q (quit) will be used to represent these operations. Repeats until the user enters the command 'q'. Prints an error message if the user enters an illegal code.

Modify *read_line* function in Laboratory 13-07-2021:

The version of *read_line* in *Chapter 13:Strings* won't work properly in the current program. Consider what happens when the user inserts a part:

Enter part number: 528

Enter part name: Disk drive

The user presses the Enter key after entering the part number, leaving an invisible new-line character that the program must read. When *scanf* reads the part number, it consumes the 5, 2, and 8, but leaves the new-line character unread. If we try to read the part name using the original *read_line* function, it will encounter the new-line character immediately and stop reading. This problem is common when numerical input is followed by character input. One solution is to write a version of *read_line* that skips white-space characters before it begins storing characters. This solves the new-line problem and also allows us to avoid storing blanks that precede the part name.

Expected output:

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10

Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2

Enter operation code: u
Enter part number: 915
Part not found.

Enter operation code: u
Enter part number: 914
Enter change in quantity on hand: +5

Enter operation code: p
Part Number   Part Name                 Quantity on Hand
    528       Disk drive                       8
    914       Printer cable                   10

Enter operation code: w
Illegal code

Enter operation code: i
Enter part number: 25
Enter part name: Printer
Enter quantity on hand: 4

Enter operation code: p
Part Number   Part Name                 Quantity on Hand
     25       Printer                          4
    528       Disk drive                       8
    914       Printer cable                   10

Enter operation code: q
```

Topic : Structures, Unions, and Enumerations

Program : Geometric Figures

Definition : The program that finds the area and perimeter(or circumference) of a geometric figure.

- Define structure types for each figure of interest including components for the figure's area and perimeter(or circumference), as well as components for those dimensions of the figure that are needed in computations of its area and perimeter.
- Then, define a union type with a component for each figure type.
- Finally, define a structure containing both a component of the union type and a component whose value denotes the correct interpretation of the union.

  Circle: area, circumference, radius (radius must be greater than zero.)
  Rectangle: area, perimeter, width, height (width and height must be greater than zero and not equal to each other.)
  Square: area, perimeter, side (side must be greater than zero.)

Operations supported by the program:

- gets the dimension data necessary to compute a figure's area and perimeter.
- computes the area of a figure given relevant dimensions.
- computes the perimeter of a figure given relevant dimensions.
- prints each component of a given object.

The program prompts the user to enter an operation code, then calls a function to perform the requested action. The codes C (circle), R (rectangle) and S (square) will be used to represent these operations. Repeats until the user enters the command 'q'.

Expected output:

```
Area and Perimeter Computation Program

Enter a letter to indicate the object shape or Q to quit.
C(circle), R(rectangle) or S(square)> c
Enter radius> 0
        The value must be greater than ZERO.
Enter radius> -1
        The value must be greater than ZERO.
Enter radius> 2
Object is a circle whose members:
        Area:12.57
        Circumference:12.57
        Radius:2.00

Enter a letter to indicate the object shape or Q to quit.
C(circle), R(rectangle) or S(square)> r
Enter height> 0
        The value must be greater than ZERO.
Enter height> -1
        The value must be greater than ZERO.
Enter height> 3
Enter width> 0
        The value must be greater than ZERO or Width and height must not equal to each other.
Enter width> -2
        The value must be greater than ZERO or Width and height must not equal to each other.
Enter width> 3
        The value must be greater than ZERO or Width and height must not equal to each other.
Enter width> 6
Object is a rectangle whose members:
        Area:18.00
        Perimeter:18.00
        Width:6.00
        Height:3.00

Enter a letter to indicate the object shape or Q to quit.
C(circle), R(rectangle) or S(square)> s
Enter length of a side> 0
        The value must be greater than ZERO.
Enter length of a side> -2
        The value must be greater than ZERO.
Enter length of a side> 3
Object is a square whose members:
        Area:9.00
        Perimeter:12.00
        Side:3.00

Enter a letter to indicate the object shape or Q to quit.
C(circle), R(rectangle) or S(square)> Q
```

Rules for Writing Quality Code

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Writing Quality Code

- Rule 1: Follow the Style Guide
- Rule 2: Create Descriptive Names
- Rule 3: Comment and Document
- Rule 4: Don't Repeat Yourself
- Rule 5: Check for Errors and Respond to Them
- Rule 6: Split Your Code into Short, Focused Units
- Rule 7: Use Framework APIs and Third-Party Libraries

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Writing Quality Code

- Rule 8: Don't Overdesign
- Rule 9: Be Consistent
- Rule 10: Avoid Security Pitfalls
- Rule 11: Use Efficient Data Structures and Algorithms
- Rule 12: Include Unit Tests
- Rule 13: Keep Your Code Portable
- Rule 14: Make Your Code Buildable
- Rule 15: Put Everything Under Version Control

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Follow the Style Guide

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Follow the Style Guide

- Every programming language has a **style guide** that tells you in great detail
  - how to **indent** your code,
  - where to **put spaces** and **braces**,
  - how to **name** stuff,
  - how to **comment**
  - all the good and bad practices.

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Create Descriptive Names

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Create Descriptive Names

- Constrained by slow, clunky teletypes, programmers **in the past** used to contract the names of their variables and routines to save time, keystrokes, ink, and paper.

- Use **long descriptive names**, like *complementSpanLength*, to help yourself, now and in the future, as well as your colleagues to understand what the code does.
  - **The only exception** to this rule concerns the few key variables used within a method's body, such as a **loop index, a parameter, an intermediate result**, or a **return value**.

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Create Descriptive Names

- There are some easy naming rules.
  - Class and type names should be nouns.
  - Methods names should contain a verb.
    - In particular, if a method returns a value indicating whether something holds true for an object, the method name should start with *is*.
    - Other methods that return an object's property should start with *get*, and those that set a property should start with *set*.

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Create Descriptive Names

- Even more importantly, think long and hard before you name something.
  - Is the name accurate?
  - Did you mean highestPrice, rather than bestPrice?
  - Is the name specific enough to avoid taking more than its fair share of semantic space?
  - Should you name your method getBestPrice, rather than getBest?
  - Does its form match that of other similar names?
  - If you have a method ReadEventLog, you shouldn't name another NetErrorLogRead.
  - If you're naming a function, does the name describe what the function returns?

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Comment and Document

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Comment and Document

- Start every routine you write (function or method) with a **comment** outlining what the routine does, its parameters, and what it returns, as well as possible errors and exceptions.
- Summarize in a comment the role of each file and class, the contents of each class field, and the major steps of complex code.
- Write the comments as you develop the code; if you think you'll add them later, you're kidding yourself.

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Comment and Document

- In addition, ensure that your code as a whole (for example, an application or library) comes with at least a guide explaining what it does;
  - indicating its dependencies;
  - and providing instructions on building, testing, installation, and use.
- This document should be short and sweet; a single README file is often enough.

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Don't Repeat Yourself

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Don't Repeat Yourself

- Never copy-and-paste code.
- Instead, abstract the common parts into a routine or class (or macro, if you must), and use it with appropriate parameters.
- More broadly, avoid duplicate instances of similar data or code.
- Keep a definitive version in one place, and let that version drive all other uses.

Check for Errors and Respond to Them

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional

Diomidis Spinellis, Code Quality: The Open Source Perspective,

Check for Errors and Respond to Them

- Routines can return with an error indication, or they can raise an exception.
- Deal with it.
- Ignoring errors and exceptions simply sweeps the problem under the carpet.

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Split Your Code into Short, Focused Units

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Split Your Code into Short, Focused Units

- Every method, function, or logical code block should fit on a reasonably-sized screen window.
  - If it's longer, split it into shorter pieces.
  - An exception can be made for simple repetitive code sequences.
- Even within a routine, divide long code sequences into blocks whose function you can describe with a comment at the beginning of each block.

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Split Your Code into Short, Focused Units

- Furthermore, each class, module, file, or process should concern one single thing.
  - If a code unit undertakes diverse responsibilities, split it accordingly.

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

SUMMARY

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Summary

- You'll eventually create code that’s
  - easier to read,
  - thoroughly tested,
  - more likely to run correctly,
  - and much simpler to revise when that time comes.
- You'll also save yourself and your program's users a lot of headaches.

Diomidis Spinellis, Code Quality: The Open Source Perspective, Addison-Wesley Professional.

Introduction

Organizing a C Program

- Major elements of a C program:
  - Preprocessing directives such as #include and #define
  - Type definitions
  - Declarations of external variables
  - Function prototypes
  - Function definitions

Organizing a C Program

- C imposes only a few rules on the order of these items:
  - A preprocessing directive doesn't take effect until the line on which it appears.
  - A type name can't be used until it's been defined.
  - A variable can't be used until it's declared.
- It's a good idea to define or declare every function prior to its first call.
  - C99 makes this a requirement.

Organizing a C Program

- There are several ways to organize a program so that these rules are obeyed.
- One possible ordering:
  - #include directives
  - #define directives
  - Type definitions
  - Declarations of external variables
  - Prototypes for functions other than main
  - Definition of main
  - Definitions of other functions

Organizing a C Program

- It's a good idea to have a boxed comment preceding each function definition.
- Information to include in the comment:
  - Name of the function
  - Purpose of the function
  - Meaning of each parameter
  - Description of return value (if any)
  - Description of side effects (such as modifying external variables)

Program: Guessing a Number

- The guess.c program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible.
- Tasks to be carried out by the program:
  - Initialize the random number generator
  - Choose a secret number
  - Interact with the user until the correct number is picked
- Each task can be handled by a separate function.

Program: Guessing a Number

```
Guess the secret number between 1 and 100.

A new number has been chosen.
Enter guess: 55
Too low; try again.
Enter guess: 65
Too high; try again.
Enter guess: 60
Too high; try again.
Enter guess: 58
You won in 4 guesses!

Play again? (Y/N) y

A new number has been chosen.
Enter guess: 78
Too high; try again.
Enter guess: 34
You won in 2 guesses!

Play again? (Y/N) n
```

Organizing a C Program: Guessing a Number

- Major elements of a C program:
  - Preprocessing directives such as #include and #define

```
/****header files****/
#include <stdio.h> // The printf() and scanf()
        functions is defined in stdio.h header file.

#include <stdlib.h> // The rand() and srand() function
        is defined in stdlib.h header file.

#include <time.h> // The time() function is defined in
        time.h header file.
```

Organizing a C Program: Guessing a Number

- Major elements of a C program:
  - Preprocessing directives such as #include and #define

```
/****constant value****/
#define MAX_NUMBER 100
```

Organizing a C Program: Guessing a Number

- Major elements of a C program:
  - Declarations of external variables

```
/****external variable****/
int secret_number;
```

Organizing a C Program : Guessing a Number

- Major elements of a C program:
  - Function prototypes

```
/****prototypes****/
void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);
```

Organizing a C Program : Guessing a Number

- Major elements of a C program:
  - Function definitions

```
/****Initializes the random number generator using the time of day.****/
void initialize_number_generator(void)
{
    srand((unsigned) time(NULL)); // generates a random number.
}
```

Organizing a C Program : Guessing a Number

- Major elements of a C program:
  - Function definitions

```
/*****Randomly selects a number between 1 and
MAX_NUMBER and stores it in secret_number****/
void choose_new_secret_number(void)
{
    secret_number = rand() % MAX_NUMBER + 1;
//selects a number
}
```

Organizing a C Program : Guessing a Number

- Major elements of a C program:
  - Function definitions

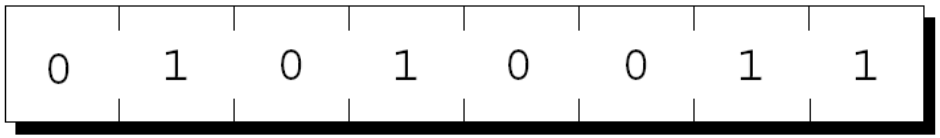
```
/*****Repeatedly reads user guesses and tells the user
whether each is too low, too high, or correct. When the
guess is correct, prints the total number of guesses and
returns.*****/
void read_guesses(void)
{
    int guess, num_guesses = 0; // local variables
    for(;;) {…}//infinite loop
}
```

Chapter 11

Pointers

Pointer Variables

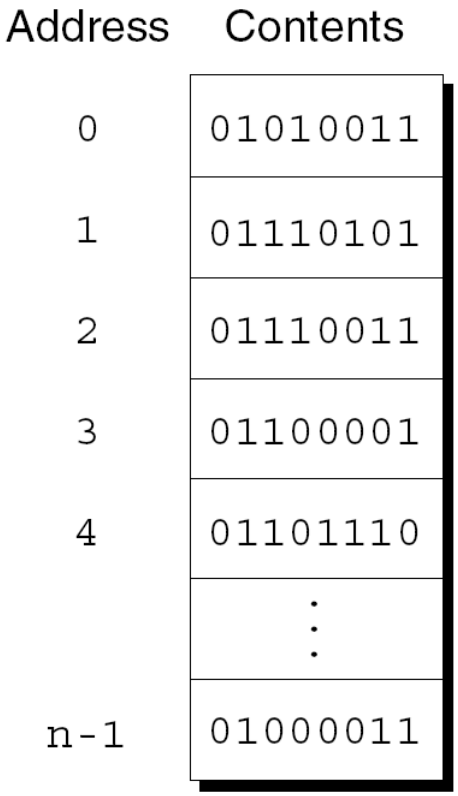
- The first step in understanding pointers is visualizing what they represent at the machine level.
- In most modern computers, main memory is divided into ***bytes,*** with each byte capable of storing eight bits of information:



- Each byte has a unique ***address.***

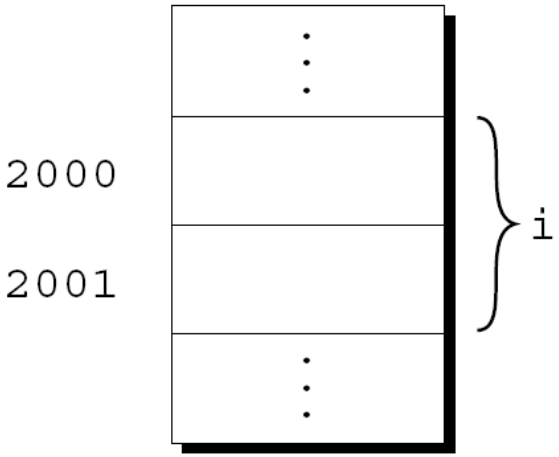
Pointer Variables

- If there are *n* bytes in memory, we can think of addresses as numbers that range from 0 to *n* – 1:



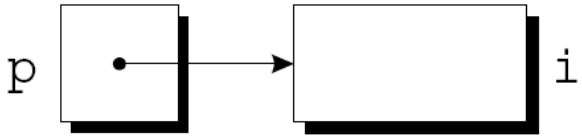
Pointer Variables

- Each variable in a program occupies one or more bytes of memory.
- The address of the first byte is said to be the address of the variable.
- In the following figure, the address of the variable `i` is 2000:



Pointer Variables

- Addresses can be stored in special ***pointer variables.***
- When we store the address of a variable `i` in the pointer variable `p`, we say that `p` "points to" `i`.
- A graphical representation:



Declaring Pointer Variables

- When a pointer variable is declared, its name must be preceded by an asterisk:

```
int *p;
```

- `p` is a pointer variable capable of pointing to ***objects*** of type `int`.
- We use the term *object* instead of *variable* since `p` might point to an area of memory that doesn't belong to a variable.

Declaring Pointer Variables

- Pointer variables can appear in declarations along with other variables:

```
int i, j, a[10], b[20], *p, *q;
```

- C requires that every pointer variable point only to objects of a particular type (the ***referenced type***):

```
int *p;      /* points only to integers   */
double *q;   /* points only to doubles    */
char *r;     /* points only to characters */
```

- There are no restrictions on what the referenced type may be.

The Address and Indirection Operators

- C provides a pair of operators designed specifically for use with pointers.
  - To find the address of a variable, we use the `&` (address) operator.
  - To gain access to the object that a pointer points to, we use the `*` (***indirection***) operator.

The Address Operator

- Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:

```
int *p;  /* points nowhere in particular */
```

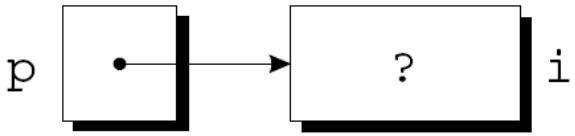
- It's crucial to initialize `p` before we use it.

The Address Operator

- One way to initialize a pointer variable is to assign it the address of a variable:

```
int i, *p;
…
p = &i;
```

- Assigning the address of `i` to the variable `p` makes `p` point to `i`:



The Address Operator

- It's also possible to initialize a pointer variable at the time it's declared:

```
int i;
int *p = &i;
```

- The declaration of `i` can even be combined with the declaration of `p`:

```
int i, *p = &i;
```

The Indirection Operator

- Once a pointer variable points to an object, we can use the `*` (indirection) operator to access what's stored in the object.
- If `p` points to `i`, we can print the value of `i` as follows:

```
printf("%d\n", *p);
```

- Applying `&` to a variable produces a pointer to the variable. Applying `*` to the pointer takes us back to the original variable:

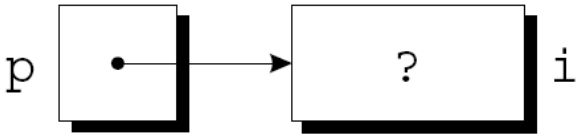
```
j = *&i;    /* same as j = i; */
```

The Indirection Operator

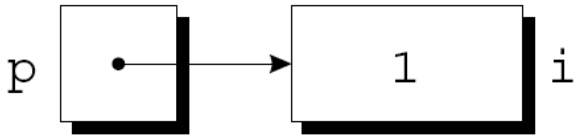
- As long as `p` points to `i`, `*p` is an ***alias*** for `i`.
  - `*p` has the same value as `i`.
  - Changing the value of `*p` changes the value of `i`.
- The example on the next slide illustrates the equivalence of `*p` and `i`.

The Indirection Operator

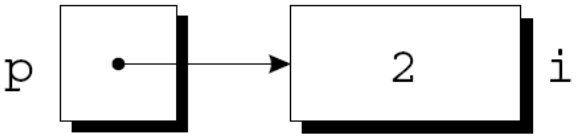
```
p = &i;
```




```
i = 1;
```




```
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
*p = 2;
```




```
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```

The Indirection Operator

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:

```
int *p;
printf("%d", *p);    /*** WRONG ***/
```

- Assigning a value to `*p` is particularly dangerous:

```
int *p;
*p = 1;    /*** WRONG ***/
```

Pointer Assignment

- C allows the use of the assignment operator to copy pointers of the same type.
- Assume that the following declaration is in effect:

```
int i, j, *p, *q;
```

- Example of pointer assignment:

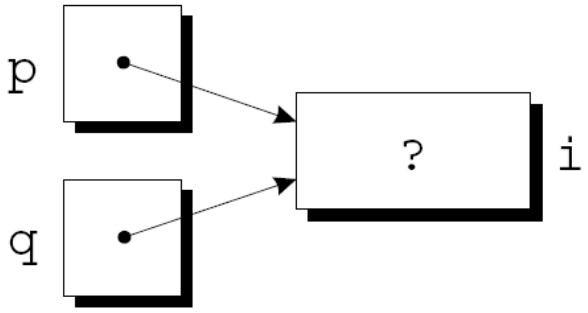
```
p = &i;
```

Pointer Assignment

- Another example of pointer assignment:

  `q = p;`

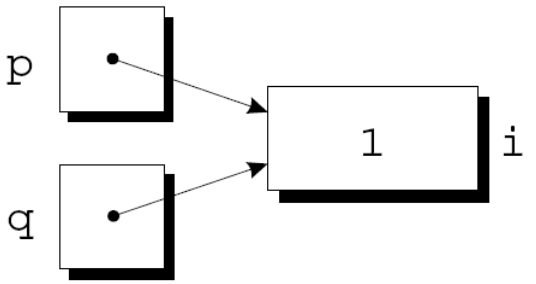
  `q` now points to the same place as `p`:



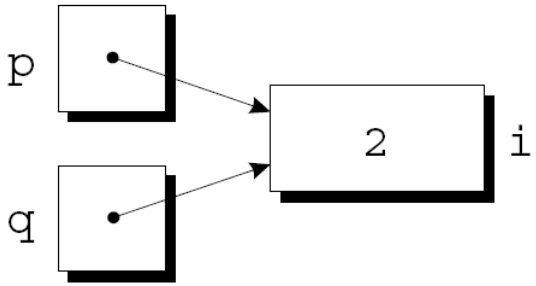
Pointer Assignment

- If `p` and `q` both point to `i`, we can change `i` by assigning a new value to either `*p` or `*q`:

```
*p = 1;
```




```
*q = 2;
```



- Any number of pointer variables may point to the same object.

Pointer Assignment

- Be careful not to confuse

  ```
  q = p;
  ```

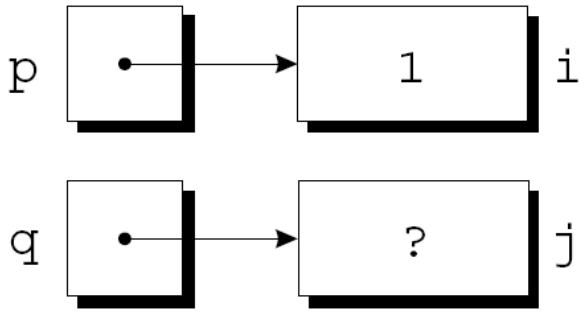
  with

  ```
  *q = *p;
  ```

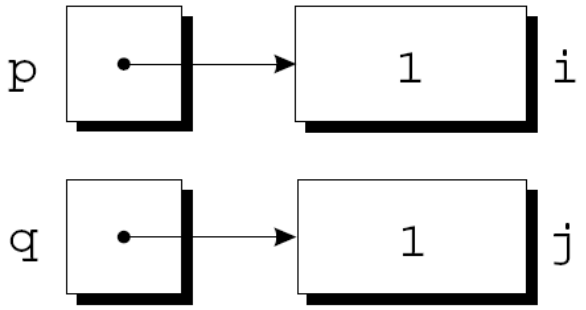
- The first statement is a pointer assignment, but the second is not.
- The example on the next slide shows the effect of the second statement.

Pointer Assignment

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



Pointers as Arguments

- In Chapter 9, we tried—and failed—to write a `decompose` function that could modify its arguments.
- By passing a *pointer* to a variable instead of the *value* of the variable, `decompose` can be fixed.

Pointers as Arguments

- New definition of `decompose`:

```
void decompose(double x, long *int_part,
               double *frac_part)
{
  *int_part = (long) x;
  *frac_part = x - *int_part;
}
```

- Possible prototypes for `decompose`:

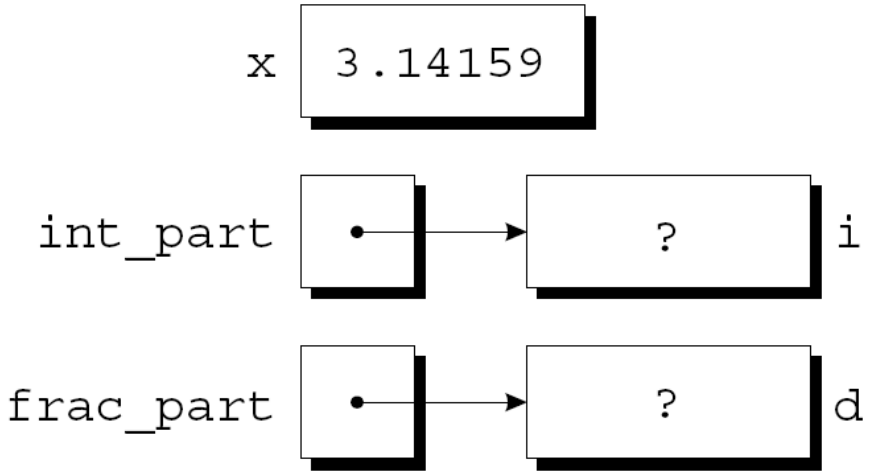
```
void decompose(double x, long *int_part,
               double *frac_part);
void decompose(double, long *, double *);
```

Pointers as Arguments

- A call of `decompose`:

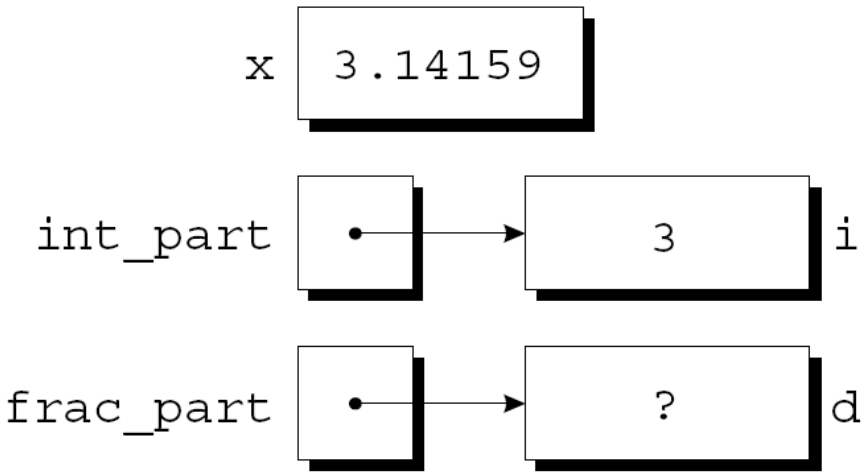
```
decompose(3.14159, &i, &d);
```

- As a result of the call, `int_part` points to `i` and `frac_part` points to `d`:



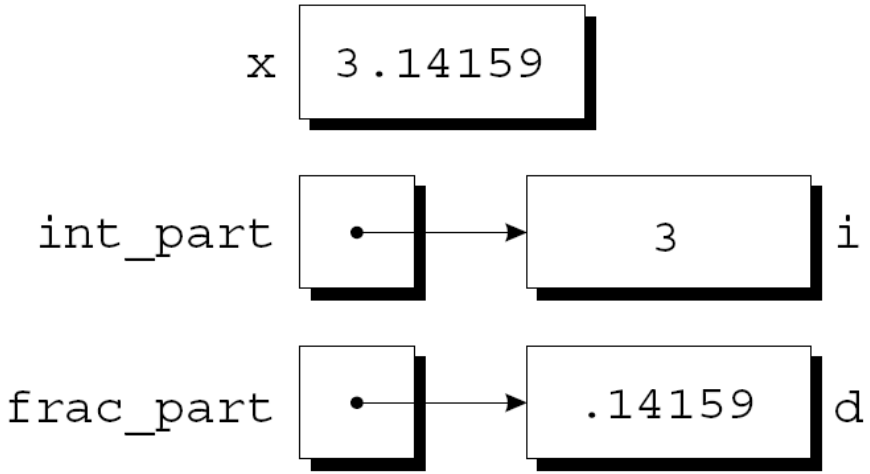
Pointers as Arguments

- The first assignment in the body of `decompose` converts the value of `x` to type `long` and stores it in the object pointed to by `int_part`:



Pointers as Arguments

- The second assignment stores `x - *int_part` into the object that `frac_part` points to:



Pointers as Arguments

- Arguments in calls of `scanf` are pointers:

```
int i;
…
scanf("%d", &i);
```

  Without the `&`, `scanf` would be supplied with the *value* of `i`.

Pointers as Arguments

- Although `scanf`'s arguments must be pointers, it's not always true that every argument needs the `&` operator:

```
int i, *p;
…
p = &i;
scanf("%d", p);
```

- Using the `&` operator in the call would be wrong:

```
scanf("%d", &p);    /*** WRONG ***/
```

Pointers as Arguments

- Failing to pass a pointer to a function when one is expected can have disastrous results.
- A call of `decompose` in which the `&` operator is missing:

```
decompose(3.14159, i, d);
```

- When `decompose` stores values in `*int_part` and `*frac_part`, it will attempt to change unknown memory locations instead of modifying `i` and `d`.
- If we've provided a prototype for `decompose`, the compiler will detect the error.
- In the case of `scanf`, however, failing to pass pointers may go undetected.

Program: Finding the Largest and Smallest Elements in an Array

- The `max_min.c` program uses a function named `max_min` to find the largest and smallest elements in an array.
- Prototype for `max_min`:

```
void max_min(int a[], int n, int *max, int *min);
```

- Example call of `max_min`:

```
max_min(b, N, &big, &small);
```

- When `max_min` finds the largest element in `b`, it stores the value in `big` by assigning it to `*max`.
- `max_min` stores the smallest element of `b` in `small` by assigning it to `*min`.

Program: Finding the Largest and Smallest Elements in an Array

- `max_min.c` will read 10 numbers into an array, pass it to the `max_min` function, and print the results:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
Largest: 102
Smallest: 7
```

maxmin.c

```
/* Finds the largest and smallest elements in an array */

#include <stdio.h>

#define N 10

void max_min(int a[], int n, int *max, int *min);

int main(void)
{
  int b[N], i, big, small;

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &b[i]);
```

```
  max_min(b, N, &big, &small);

  printf("Largest: %d\n", big);
  printf("Smallest: %d\n", small);

  return 0;
}

void max_min(int a[], int n, int *max, int *min)
{
  int i;

  *max = *min = a[0];
  for (i = 1; i < n; i++) {
    if (a[i] > *max)
      *max = a[i];
    else if (a[i] < *min)
      *min = a[i];
  }
}
```

Using `const` to Protect Arguments

- When an argument is a pointer to a variable `x`, we normally assume that `x` will be modified:

```
f(&x);
```

- It's possible, though, that `f` merely needs to examine the value of `x`, not change it.
- The reason for the pointer might be efficiency: passing the value of a variable can waste time and space if the variable requires a large amount of storage.

Using `const` to Protect Arguments

- We can use `const` to document that a function won't change an object whose address is passed to the function.
- `const` goes in the parameter's declaration, just before the specification of its type:

```
void f(const int *p)
{
  *p = 0;    /*** WRONG ***/
}
```

Attempting to modify `*p` is an error that the compiler will detect.

Pointers as Return Values

- Functions are allowed to return pointers:

```
int *max(int *a, int *b)
{
  if (*a > *b)
    return a;
  else
    return b;
}
```

- A call of the `max` function:

```
int *p, i, j;
…
p = max(&i, &j);
```

After the call, `p` points to either `i` or `j`.

Pointers as Return Values

- Although `max` returns one of the pointers passed to it as an argument, that's not the only possibility.
- A function could also return a pointer to an external variable or to a static local variable.
- Never return a pointer to an *automatic* local variable:

```
int *f(void)
{
  int i;
  …
  return &i;
}
```

The variable `i` won't exist after `f` returns.

Pointers as Return Values

- Pointers can point to array elements.
- If `a` is an array, then `&a[i]` is a pointer to element `i` of `a`.
- It's sometimes useful for a function to return a pointer to one of the elements in an array.
- A function that returns a pointer to the middle element of `a`, assuming that `a` has `n` elements:

```
int *find_middle(int a[], int n) {
  return &a[n/2];
}
```

Chapter 13

Strings

Introduction

- This chapter covers both string *constants* (or *literals*, as they're called in the C standard) and string *variables*.
- Strings are arrays of characters in which a special character—the null character—marks the end.
- The C library provides a collection of functions for working with strings.

String Literals

- A ***string literal*** is a sequence of characters enclosed within double quotes:

  ```
  "When you come to a fork in the road, take it."
  ```

- String literals may contain escape sequences.
- Character escapes often appear in `printf` and `scanf` format strings.
- For example, each `\n` character in the string

  ```
  "Candy\nIs dandy\nBut liquor\nIs quicker.\n  --Ogden Nash\n"
  ```

  causes the cursor to advance to the next line:

  ```
  Candy
  Is dandy
  But liquor
  Is quicker.
    --Ogden Nash
  ```

Continuing a String Literal

- The backslash character (\) can be used to continue a string literal from one line to the next:

```
printf("When you come to a fork in the road, take it.  \
--Yogi Berra");
```

- In general, the \ character can be used to join two or more lines of a program into a single line.

Continuing a String Literal

- There's a better way to deal with long string literals.
- When two or more string literals are adjacent, the compiler will join them into a single string.
- This rule allows us to split a string literal over two or more lines:

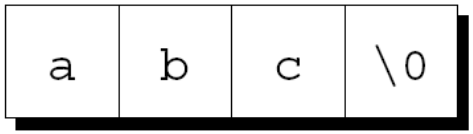
```
printf("When you come to a fork in the road, take it.  "
       "--Yogi Berra");
```

How String Literals Are Stored

- When a C compiler encounters a string literal of length *n* in a program, it sets aside *n* + 1 bytes of memory for the string.
- This memory will contain the characters in the string, plus one extra character—the ***null character***—to mark the end of the string.
- The null character is a byte whose bits are all zero, so it's represented by the `\0` escape sequence.

How String Literals Are Stored

- The string literal `"abc"` is stored as an array of four characters:



- The string `""` is stored as a single null character:

How String Literals Are Stored

- Since a string literal is stored as an array, the compiler treats it as a pointer of type `char *`.
- Both `printf` and `scanf` expect a value of type `char *` as their first argument.
- The following call of `printf` passes the address of `"abc"` (a pointer to where the letter `a` is stored in memory):

```
printf("abc");
```

Operations on String Literals

- We can use a string literal wherever C allows a `char *` pointer:

```
char *p;

p = "abc";
```

- This assignment makes `p` point to the first character of the string.

Operations on String Literals

- String literals can be subscripted:

```
char ch;

ch = "abc"[1];
```

  The new value of `ch` will be the letter `b`.

- A function that converts a number between 0 and 15 into the equivalent hex digit:

```
char digit_to_hex_char(int digit)
{
  return "0123456789ABCDEF"[digit];
}
```

Operations on String Literals

- Attempting to modify a string literal causes undefined behavior:

```
char *p = "abc";

*p = 'd';    /*** WRONG ***/
```

- A program that tries to change a string literal may crash or behave erratically.

String Literals versus Character Constants

- A string literal containing a single character isn't the same as a character constant.
  - `"a"` is represented by a *pointer*.
  - `'a'` is represented by an *integer*.
- A legal call of `printf`:

```
printf("\n");
```

- An illegal call:

```
printf('\n');    /*** WRONG ***/
```

String Variables

- Any one-dimensional array of characters can be used to store a string.
- A string must be terminated by a null character.
- Difficulties with this approach:
  - It can be hard to tell whether an array of characters is being used as a string.
  - String-handling functions must be careful to deal properly with the null character.
  - Finding the length of a string requires searching for the null character.

String Variables

- If a string variable needs to hold 80 characters, it must be declared to have length 81:

```
#define STR_LEN 80
…
char str[STR_LEN+1];
```

- Adding 1 to the desired length allows room for the null character at the end of the string.
- Defining a macro that represents 80 and then adding 1 separately is a common practice.

String Variables

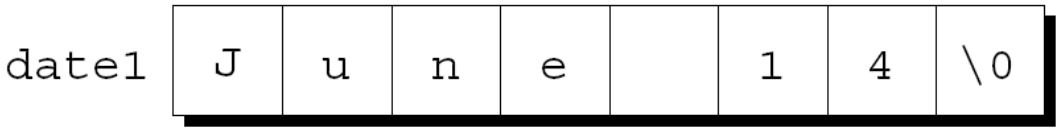
- Be sure to leave room for the null character when declaring a string variable.
- Failing to do so may cause unpredictable results when the program is executed.
- The actual length of a string depends on the position of the terminating null character.
- An array of `STR_LEN` + 1 characters can hold strings with lengths between 0 and `STR_LEN`.

Initializing a String Variable

- A string variable can be initialized at the same time it's declared:

```
char date1[8] = "June 14";
```

- The compiler will automatically add a null character so that `date1` can be used as a string:



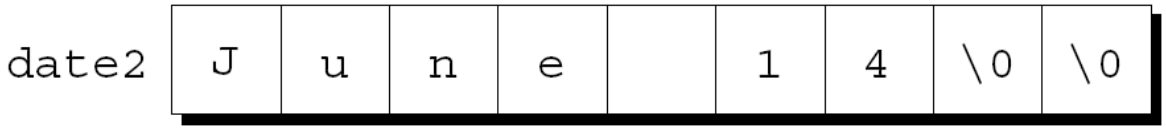
- `"June 14"` is not a string literal in this context.
- Instead, C views it as an abbreviation for an array initializer.

Initializing a String Variable

- If the initializer is too short to fill the string variable, the compiler adds extra null characters:

```
char date2[9] = "June 14";
```

Appearance of `date2`:

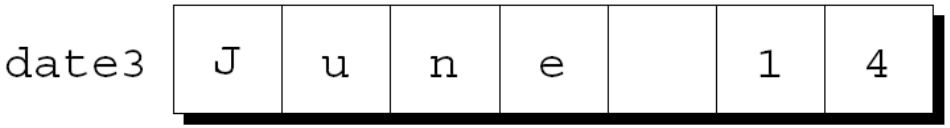


Initializing a String Variable

- An initializer for a string variable can't be longer than the variable, but it can be the same length:

```
char date3[7] = "June 14";
```

- There's no room for the null character, so the compiler makes no attempt to store one:



Initializing a String Variable

- The declaration of a string variable may omit its length, in which case the compiler computes it:

```
char date4[] = "June 14";
```

- The compiler sets aside eight characters for `date4`, enough to store the characters in `"June 14"` plus a null character.
- Omitting the length of a string variable is especially useful if the initializer is long, since computing the length by hand is error-prone.

Character Arrays versus Character Pointers

- The declaration

  ```
  char date[] = "June 14";
  ```

  declares `date` to be an *array*,
- The similar-looking

  ```
  char *date = "June 14";
  ```

  declares `date` to be a *pointer*.
- Thanks to the close relationship between arrays and pointers, either version can be used as a string.

Character Arrays versus Character Pointers

- However, there are significant differences between the two versions of `date`.
  - In the array version, the characters stored in `date` can be modified. In the pointer version, `date` points to a string literal that shouldn't be modified.
  - In the array version, `date` is an array name. In the pointer version, `date` is a variable that can point to other strings.

Character Arrays versus Character Pointers

- The declaration

  ```
  char *p;
  ```

  does not allocate space for a string.
- Before we can use `p` as a string, it must point to an array of characters.
- One possibility is to make `p` point to a string variable:

  ```
  char str[STR_LEN+1], *p;

  p = str;
  ```
- Another possibility is to make `p` point to a dynamically allocated string.

Character Arrays versus Character Pointers

- Using an uninitialized pointer variable as a string is a serious error.
- An attempt at building the string `"abc"`:

```
char *p;

p[0] = 'a';     /*** WRONG ***/
p[1] = 'b';     /*** WRONG ***/
p[2] = 'c';     /*** WRONG ***/
p[3] = '\0';    /*** WRONG ***/
```

- Since `p` hasn't been initialized, this causes undefined behavior.

Reading and Writing Strings

- Writing a string is easy using either `printf` or `puts`.
- Reading a string is a bit harder, because the input may be longer than the string variable into which it's being stored.
- To read a string in a single step, we can use either `scanf` or `gets`.
- As an alternative, we can read strings one character at a time.

Writing Strings Using `printf` and `puts`

- The `%s` conversion specification allows `printf` to write a string:

```
char str[] = "Are we having fun yet?";

printf("%s\n", str);
```

  The output will be

```
Are we having fun yet?
```

- `printf` writes the characters in a string one by one until it encounters a null character.

Writing Strings Using `printf` and `puts`

- To print part of a string, use the conversion specification `%.`*p*`s`.
- *p* is the number of characters to be displayed.
- The statement

```
printf("%.6s\n", str);
```

will print

```
Are we
```

Writing Strings Using `printf` and `puts`

- The `%`*m*`s` conversion will display a string in a field of size *m*.
- If the string has fewer than *m* characters, it will be right-justified within the field.
- To force left justification instead, we can put a minus sign in front of *m*.
- The *m* and *p* values can be used in combination.
- A conversion specification of the form `%`*m*`.`*p*`s` causes the first *p* characters of a string to be displayed in a field of size *m*.

Writing Strings Using `printf` and `puts`

- `printf` isn't the only function that can write strings.
- The C library also provides `puts`:

```
puts(str);
```

- After writing a string, `puts` always writes an additional new-line character.

Reading Strings Using `scanf` and `gets`

- The `%s` conversion specification allows `scanf` to read a string into a character array:

```
scanf("%s", str);
```

- `str` is treated as a pointer, so there's no need to put the `&` operator in front of `str`.
- When `scanf` is called, it skips white space, then reads characters and stores them in `str` until it encounters a white-space character.
- `scanf` always stores a null character at the end of the string.

Reading Strings Using `scanf` and `gets`

- `scanf` won't usually read a full line of input.
- A new-line character will cause `scanf` to stop reading, but so will a space or tab character.
- To read an entire line of input, we can use `gets`.
- Properties of `gets`:
  - Doesn't skip white space before starting to read input.
  - Reads until it finds a new-line character.
  - Discards the new-line character instead of storing it; the null character takes its place.

Reading Strings Using `scanf` and `gets`

- Consider the following program fragment:

```
char sentence[SENT_LEN+1];

printf("Enter a sentence:\n");
scanf("%s", sentence);
```

- Suppose that after the prompt

```
Enter a sentence:
```

the user enters the line

```
To C, or not to C: that is the question.
```

- `scanf` will store the string `"To"` in `sentence`.

Reading Strings Using `scanf` and `gets`

- Suppose that we replace `scanf` by `gets`:

  ```
  gets(sentence);
  ```

- When the user enters the same input as before, `gets` will store the string

  ```
  " To C, or not to C: that is the question."
  ```

  in `sentence`.

Reading Strings Using `scanf` and `gets`

- As they read characters into an array, `scanf` and `gets` have no way to detect when it's full.
- Consequently, they may store characters past the end of the array, causing undefined behavior.
- `scanf` can be made safer by using the conversion specification `%`*n*`s` instead of `%s`.
- *n* is an integer indicating the maximum number of characters to be stored.
- `gets` is inherently unsafe; `fgets` is a much better alternative.

Reading Strings Character by Character

- Programmers often write their own input functions.
- Issues to consider:
  - Should the function skip white space before beginning to store the string?
  - What character causes the function to stop reading: a new-line character, any white-space character, or some other character? Is this character stored in the string or discarded?
  - What should the function do if the input string is too long to store: discard the extra characters or leave them for the next input operation?

Reading Strings Character by Character

- Suppose we need a function that (1) doesn't skip white-space characters, (2) stops reading at the first new-line character (which isn't stored in the string), and (3) discards extra characters.
- A prototype for the function:

```
int read_line(char str[], int n);
```

- If the input line contains more than `n` characters, `read_line` will discard the additional characters.
- `read_line` will return the number of characters it stores in `str`.

Reading Strings Character by Character

- `read_line` consists primarily of a loop that calls `getchar` to read a character and then stores the character in `str`, provided that there's room left:

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';    /* terminates string */
  return i;         /* number of characters stored */
}
```

- `ch` has `int` type rather than `char` type because `getchar` returns an `int` value.

Reading Strings Character by Character

- Before returning, `read_line` puts a null character at the end of the string.
- Standard functions such as `scanf` and `gets` automatically put a null character at the end of an input string.
- If we're writing our own input function, we must take on that responsibility.

Accessing the Characters in a String

- Since strings are stored as arrays, we can use subscripting to access the characters in a string.
- To process every character in a string `s`, we can set up a loop that increments a counter `i` and selects characters via the expression `s[i]`.

Accessing the Characters in a String

- A function that counts the number of spaces in a string:

```
int count_spaces(const char s[])
{
  int count = 0, i;

  for (i = 0; s[i] != '\0'; i++)
    if (s[i] == ' ')
      count++;
  return count;
}
```

Accessing the Characters in a String

- A version that uses pointer arithmetic instead of array subscripting :

```
int count_spaces(const char *s)
{
  int count = 0;

  for (; *s != '\0'; s++)
    if (*s == ' ')
      count++;
  return count;
}
```

Accessing the Characters in a String

- Questions raised by the `count_spaces` example:
  - *Is it better to use array operations or pointer operations to access the characters in a string?* We can use either or both. Traditionally, C programmers lean toward using pointer operations.
  - *Should a string parameter be declared as an array or as a pointer?* There's no difference between the two.
  - *Does the form of the parameter (`s[]` or `*s`) affect what can be supplied as an argument?* No.

Using the C String Library

- Some programming languages provide operators that can copy strings, compare strings, concatenate strings, select substrings, and the like.
- C's operators, in contrast, are essentially useless for working with strings.
- Strings are treated as arrays in C, so they're restricted in the same ways as arrays.
- In particular, they can't be copied or compared using operators.

Using the C String Library

- Direct attempts to copy or compare strings will fail.
- Copying a string into a character array using the = operator is not possible:

```
char str1[10], str2[10];
…
str1 = "abc";   /*** WRONG ***/
str2 = str1;    /*** WRONG ***/
```

  Using an array name as the left operand of = is illegal.
- *Initializing* a character array using = is legal, though:

```
char str1[10] = "abc";
```

  In this context, = is not the assignment operator.

Using the C String Library

- Attempting to compare strings using a relational or equality operator is legal but won't produce the desired result:

```
if (str1 == str2) …   /*** WRONG ***/
```

- This statement compares `str1` and `str2` as *pointers*.
- Since `str1` and `str2` have different addresses, the expression `str1 == str2` must have the value 0.

Using the C String Library

- The C library provides a rich set of functions for performing operations on strings.
- Programs that need string operations should contain the following line:

```
#include <string.h>
```

- In subsequent examples, assume that `str1` and `str2` are character arrays used as strings.

The `strcpy` (String Copy) Function

- Prototype for the `strcpy` function:

```
char *strcpy(char *s1, const char *s2);
```

- `strcpy` copies the string `s2` into the string `s1`.
  - To be precise, we should say "`strcpy` copies the string pointed to by `s2` into the array pointed to by `s1`."
- `strcpy` returns `s1` (a pointer to the destination string).

The `strcpy` (String Copy) Function

- A call of `strcpy` that stores the string `"abcd"` in `str2`:

```
strcpy(str2, "abcd");
  /* str2 now contains "abcd" */
```

- A call that copies the contents of `str2` into `str1`:

```
strcpy(str1, str2);
  /* str1 now contains "abcd" */
```

The `strcpy` (String Copy) Function

- In the call `strcpy(str1, str2)`, `strcpy` has no way to check that the `str2` string will fit in the array pointed to by `str1`.
- If it doesn't, undefined behavior occurs.

The `strcpy` (String Copy) Function

- Calling the `strncpy` function is a safer, albeit slower, way to copy a string.
- `strncpy` has a third argument that limits the number of characters that will be copied.
- A call of `strncpy` that copies `str2` into `str1`:

```
strncpy(str1, str2, sizeof(str1));
```

The `strcpy` (String Copy) Function

- `strncpy` will leave `str1` without a terminating null character if the length of `str2` is greater than or equal to the size of the `str1` array.
- A safer way to use `strncpy`:

```
strncpy(str1, str2, sizeof(str1) - 1);
str1[sizeof(str1)-1] = '\0';
```

- The second statement guarantees that `str1` is always null-terminated.

The `strlen` (String Length) Function

- Prototype for the `strlen` function:

```
size_t strlen(const char *s);
```

- `size_t` is a `typedef` name that represents one of C's unsigned integer types.

The `strlen` (String Length) Function

- `strlen` returns the length of a string `s`, not including the null character.
- Examples:

```
int len;

len = strlen("abc");  /* len is now 3 */
len = strlen("");     /* len is now 0 */
strcpy(str1, "abc");
len = strlen(str1);   /* len is now 3 */
```

The `strcat` (String Concatenation) Function

- Prototype for the `strcat` function:

```
char *strcat(char *s1, const char *s2);
```

- `strcat` appends the contents of the string `s2` to the end of the string `s1`.
- It returns `s1` (a pointer to the resulting string).
- `strcat` examples:

```
strcpy(str1, "abc");
strcat(str1, "def");
  /* str1 now contains "abcdef" */
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);
  /* str1 now contains "abcdef" */
```

The `strcat` (String Concatenation) Function

- As with `strcpy`, the value returned by `strcat` is normally discarded.
- The following example shows how the return value might be used:

```
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, strcat(str2, "ghi"));
  /* str1 now contains "abcdefghi";
     str2 contains "defghi" */
```

The `strcat` (String Concatenation) Function

- `strcat(str1, str2)` causes undefined behavior if the `str1` array isn't long enough to accommodate the characters from `str2`.
- Example:

```
char str1[6] = "abc";

strcat(str1, "def");   /*** WRONG ***/
```

- `str1` is limited to six characters, causing `strcat` to write past the end of the array.

The `strcat` (String Concatenation) Function

- The `strncat` function is a safer but slower version of `strcat`.
- Like `strncpy`, it has a third argument that limits the number of characters it will copy.
- A call of `strncat`:

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

- `strncat` will terminate `str1` with a null character, which isn't included in the third argument.

The `strcmp` (String Comparison) Function

- Prototype for the `strcmp` function:

```
int strcmp(const char *s1, const char *s2);
```

- `strcmp` compares the strings `s1` and `s2`, returning a value less than, equal to, or greater than 0, depending on whether `s1` is less than, equal to, or greater than `s2`.

The `strcmp` (String Comparison) Function

- Testing whether `str1` is less than `str2`:

```
if (strcmp(str1, str2) < 0)   /* is str1 < str2? */
  …
```

- Testing whether `str1` is less than or equal to `str2`:

```
if (strcmp(str1, str2) <= 0) /* is str1 <= str2? */
  …
```

- By choosing the proper operator (<, <=, >, >=, ==, !=), we can test any possible relationship between `str1` and `str2`.

The `strcmp` (String Comparison) Function

- `strcmp` considers `s1` to be less than `s2` if either one of the following conditions is satisfied:
  - The first *i* characters of `s1` and `s2` match, but the (*i*+1)st character of `s1` is less than the (*i*+1)st character of `s2`.
  - All characters of `s1` match `s2`, but `s1` is shorter than `s2`.

The `strcmp` (String Comparison) Function

- As it compares two strings, `strcmp` looks at the numerical codes for the characters in the strings.
- Some knowledge of the underlying character set is helpful to predict what `strcmp` will do.
- Important properties of ASCII:
  - A–Z, a–z, and 0–9 have consecutive codes.
  - All upper-case letters are less than all lower-case letters.
  - Digits are less than letters.
  - Spaces are less than all printing characters.

Program: Printing a One-Month Reminder List

- The `remind.c` program prints a one-month list of daily reminders.
- The user will enter a series of reminders, with each prefixed by a day of the month.
- When the user enters 0 instead of a valid day, the program will print a list of all reminders entered, sorted by day.
- The next slide shows a session with the program.

Program: Printing a One-Month Reminder List

```
Enter day and reminder: 24 Susan's birthday
Enter day and reminder: 5 6:00 - Dinner with Marge and Russ
Enter day and reminder: 26 Movie - "Chinatown"
Enter day and reminder: 7 10:30 - Dental appointment
Enter day and reminder: 12 Movie - "Dazed and Confused"
Enter day and reminder: 5 Saturday class
Enter day and reminder: 12 Saturday class
Enter day and reminder: 0

Day Reminder
  5 Saturday class
  5 6:00 - Dinner with Marge and Russ
  7 10:30 - Dental appointment
 12 Saturday class
 12 Movie - "Dazed and Confused"
 24 Susan's birthday
 26 Movie - "Chinatown"
```

Program: Printing a One-Month Reminder List

- Overall strategy:
  - Read a series of day-and-reminder combinations.
  - Store them in order (sorted by day).
  - Display them.
- `scanf` will be used to read the days.
- `read_line` will be used to read the reminders.

Program: Printing a One-Month Reminder List

- The strings will be stored in a two-dimensional array of characters.
- Each row of the array contains one string.
- Actions taken after the program reads a day and its associated reminder:
  - Search the array to determine where the day belongs, using `strcmp` to do comparisons.
  - Use `strcpy` to move all strings below that point down one position.
  - Copy the day into the array and call `strcat` to append the reminder to the day.

Program: Printing a One-Month Reminder List

- One complication: how to right-justify the days in a two-character field.
- A solution: use `scanf` to read the day into an integer variable, than call `sprintf` to convert the day back into string form.
- `sprintf` is similar to `printf`, except that it writes output into a string.
- The call

  ```
  sprintf(day_str, "%2d", day);
  ```

  writes the value of `day` into `day_str`.

Program: Printing a One-Month Reminder List

- The following call of `scanf` ensures that the user doesn't enter more than two digits:

```
scanf("%2d", &day);
```

remind.c

```
/* Prints a one-month reminder list */

#include <stdio.h>
#include <string.h>

#define MAX_REMIND 50   /* maximum number of reminders */
#define MSG_LEN 60      /* max length of reminder message */

int read_line(char str[], int n);

int main(void)
{
  char reminders[MAX_REMIND][MSG_LEN+3];
  char day_str[3], msg_str[MSG_LEN+1];
  int day, i, j, num_remind = 0;

  for (;;) {
    if (num_remind == MAX_REMIND) {
      printf("-- No space left --\n");
      break;
    }
```

```
    printf("Enter day and reminder: ");
    scanf("%2d", &day);
    if (day == 0)
      break;
    sprintf(day_str, "%2d", day);
    read_line(msg_str, MSG_LEN);

    for (i = 0; i < num_remind; i++)
      if (strcmp(day_str, reminders[i]) < 0)
        break;
    for (j = num_remind; j > i; j--)
      strcpy(reminders[j], reminders[j-1]);

    strcpy(reminders[i], day_str);
    strcat(reminders[i], msg_str);

    num_remind++;
  }

  printf("\nDay Reminder\n");
  for (i = 0; i < num_remind; i++)
    printf(" %s\n", reminders[i]);

  return 0;
}
```

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```

String Idioms

- Functions that manipulate strings are a rich source of idioms.
- We'll explore some of the most famous idioms by using them to write the `strlen` and `strcat` functions.

Searching for the End of a String

- A version of `strlen` that searches for the end of a string, using a variable to keep track of the string's length:

```
size_t strlen(const char *s)
{
  size_t n;

  for (n = 0; *s != '\0'; s++)
    n++;
  return n;
}
```

Searching for the End of a String

- To condense the function, we can move the initialization of `n` to its declaration:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s != '\0'; s++)
    n++;
  return n;
}
```

Searching for the End of a String

- The condition `*s != '\0'` is the same as `*s != 0`, which in turn is the same as `*s`.
- A version of `strlen` that uses these observations:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s; s++)
    n++;
  return n;
}
```

Searching for the End of a String

- The next version increments `s` and tests `*s` in the same expression:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  for (; *s++;)
    n++;
  return n;
}
```

Searching for the End of a String

- Replacing the `for` statement with a `while` statement gives the following version of `strlen`:

```
size_t strlen(const char *s)
{
  size_t n = 0;

  while (*s++)
    n++;
  return n;
}
```

Searching for the End of a String

- Although we've condensed `strlen` quite a bit, it's likely that we haven't increased its speed.
- A version that *does* run faster, at least with some compilers:

```
size_t strlen(const char *s)
{
  const char *p = s;

  while (*s)
    s++;
  return s - p;
}
```

Searching for the End of a String

- Idioms for "search for the null character at the end of a string":

```
while (*s)          while (*s++)
  s++;                ;
```

- The first version leaves `s` pointing to the null character.
- The second version is more concise, but leaves `s` pointing just past the null character.

Copying a String

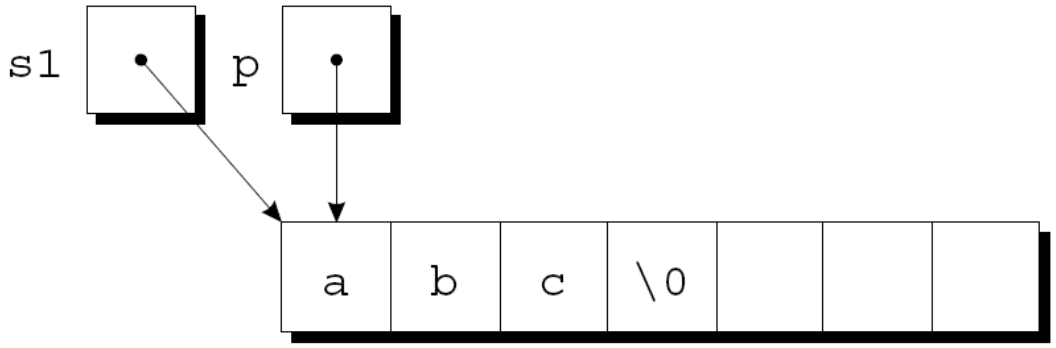
- Copying a string is another common operation.
- To introduce C's "string copy" idiom, we'll develop two versions of the `strcat` function.
- The first version of `strcat` (next slide) uses a two-step algorithm:
  - Locate the null character at the end of the string `s1` and make `p` point to it.
  - Copy characters one by one from `s2` to where `p` is pointing.

Copying a String

```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while (*p != '\0')
    p++;
  while (*s2 != '\0') {
    *p = *s2;
    p++;
    s2++;
  }
  *p = '\0';
  return s1;
}
```

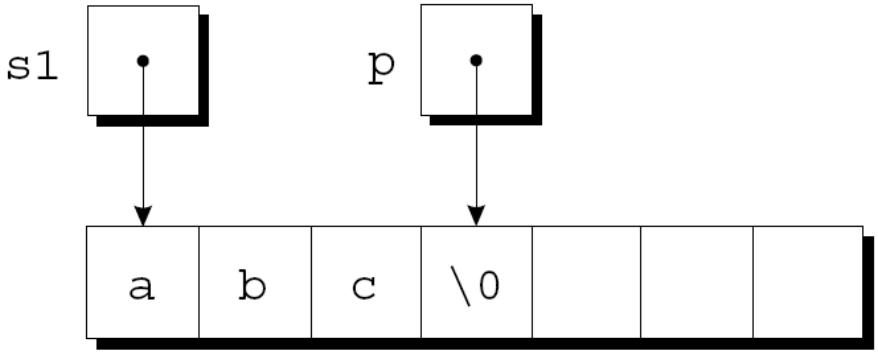
Copying a String

- `p` initially points to the first character in the `s1` string:



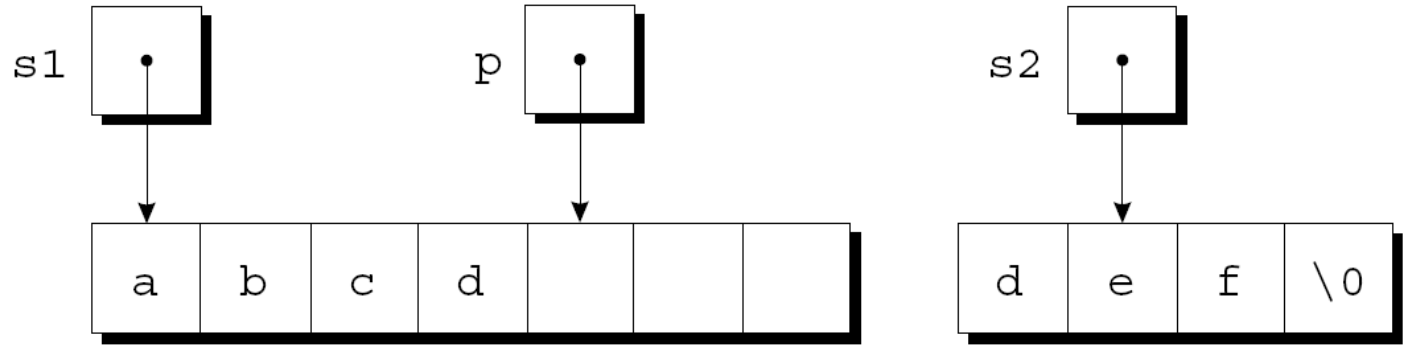
Copying a String

- The first `while` statement locates the null character at the end of `s1` and makes `p` point to it:



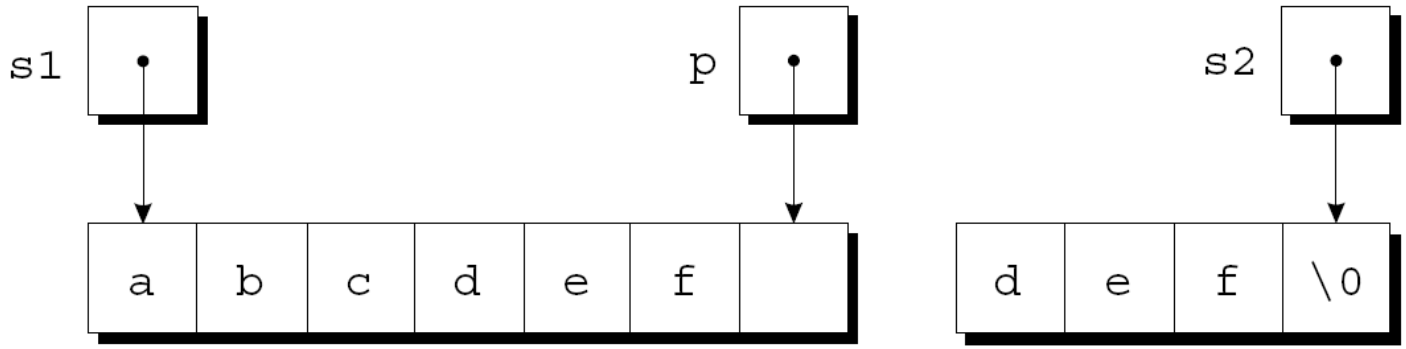
Copying a String

- The second `while` statement repeatedly copies one character from where `s2` points to where `p` points, then increments both `p` and `s2`.
- Assume that `s2` originally points to the string `"def"`.
- The strings after the first loop iteration:



Copying a String

- The loop terminates when `s2` points to the null character:



- After putting a null character where `p` is pointing, `strcat` returns.

Copying a String

- Condensed version of `strcat`:

```
char *strcat(char *s1, const char *s2)
{
  char *p = s1;

  while (*p)
    p++;
  while (*p++ = *s2++)
    ;
  return s1;
}
```

Copying a String

- The heart of the streamlined `strcat` function is the "string copy" idiom:

```
while (*p++ = *s2++)
  ;
```

- Ignoring the two `++` operators, the expression inside the parentheses is an assignment:

```
*p = *s2
```

- After the assignment, `p` and `s2` are incremented.
- Repeatedly evaluating this expression copies characters from where `s2` points to where `p` points.

Copying a String

- But what causes the loop to terminate?
- The `while` statement tests the character that was copied by the assignment `*p = *s2`.
- All characters except the null character test true.
- The loop terminates *after* the assignment, so the null character will be copied.

Arrays of Strings

- There is more than one way to store an array of strings.
- One option is to use a two-dimensional array of characters, with one string per row:

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

- The number of rows in the array can be omitted, but we must specify the number of columns.

Arrays of Strings

- Unfortunately, the `planets` array contains a fair bit of wasted space (extra null characters):

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

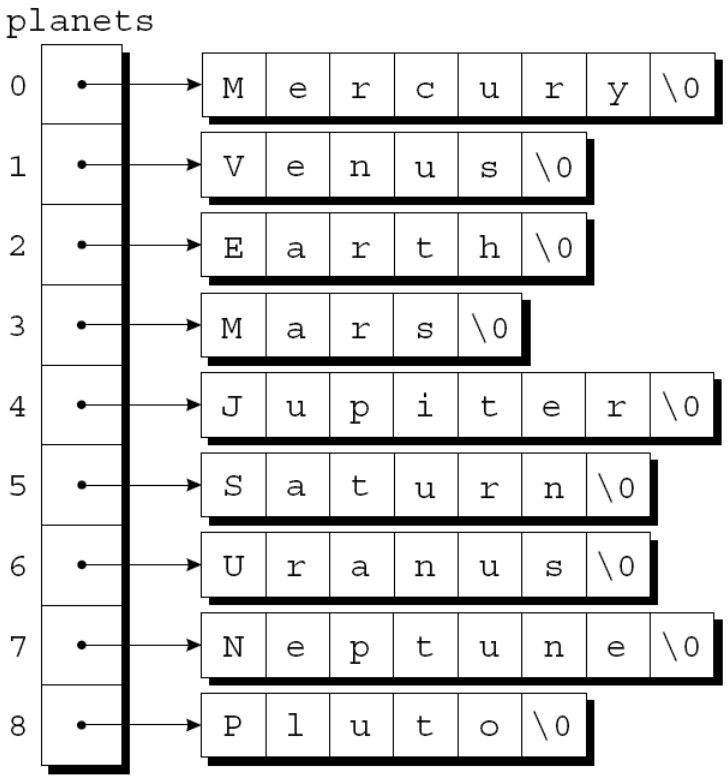
Arrays of Strings

- Most collections of strings will have a mixture of long strings and short strings.
- What we need is a ***ragged array,*** whose rows can have different lengths.
- We can simulate a ragged array in C by creating an array whose elements are *pointers* to strings:

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

Arrays of Strings

- This small change has a dramatic effect on how `planets` is stored:



Arrays of Strings

- To access one of the planet names, all we need do is subscript the `planets` array.
- Accessing a character in a planet name is done in the same way as accessing an element of a two-dimensional array.
- A loop that searches the `planets` array for strings beginning with the letter M:

```
for (i = 0; i < 9; i++)
  if (planets[i][0] == 'M')
    printf("%s begins with M\n", planets[i]);
```

Command-Line Arguments

- When we run a program, we'll often need to supply it with information.
- This may include a file name or a switch that modifies the program's behavior.
- Examples of the UNIX `ls` command:

```
ls
ls -l
ls -l remind.c
```

Command-Line Arguments

- Command-line information is available to all programs, not just operating system commands.
- To obtain access to ***command-line arguments***, `main` must have two parameters:

```
int main(int argc, char *argv[])
{
  …
}
```

- Command-line arguments are called ***program parameters*** in the C standard.

Command-Line Arguments

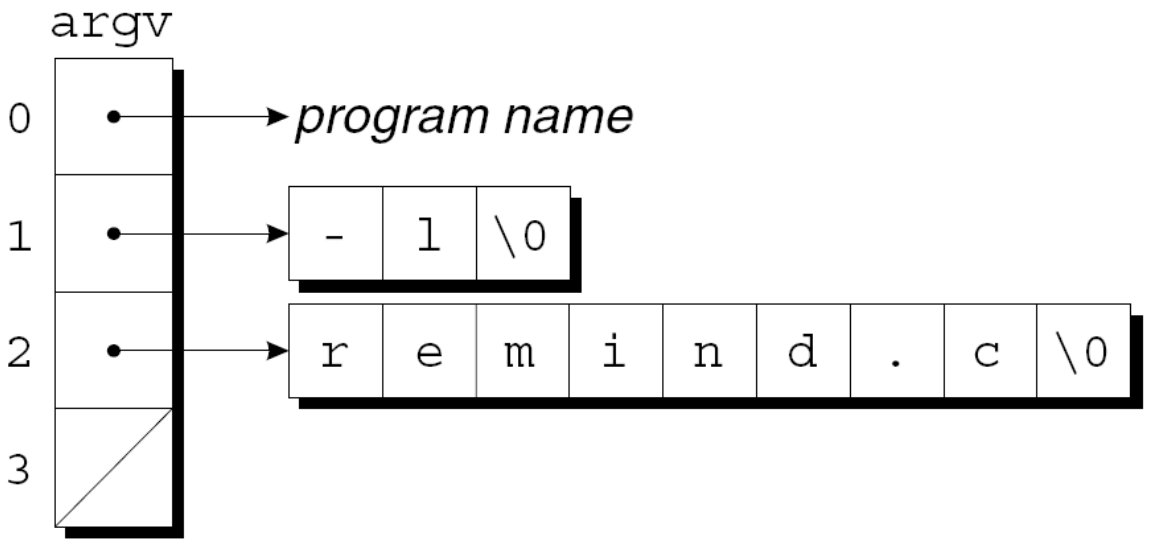
- `argc` ("argument count") is the number of command-line arguments.
- `argv` ("argument vector") is an array of pointers to the command-line arguments (stored as strings).
- `argv[0]` points to the name of the program, while `argv[1]` through `argv[argc-1]` point to the remaining command-line arguments.
- `argv[argc]` is always a ***null pointer***—a special pointer that points to nothing.
  - The macro `NULL` represents a null pointer.

Command-Line Arguments

- If the user enters the command line

  ```
  ls -l remind.c
  ```

  then `argc` will be 3, and `argv` will have the following appearance:



Command-Line Arguments

- Since `argv` is an array of pointers, accessing command-line arguments is easy.
- Typically, a program that expects command-line arguments will set up a loop that examines each argument in turn.
- One way to write such a loop is to use an integer variable as an index into the `argv` array:

```
int i;

for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

Command-Line Arguments

- Another technique is to set up a pointer to `argv[1]`, then increment the pointer repeatedly:

```
char **p;

for (p = &argv[1]; *p != NULL; p++)
  printf("%s\n", *p);
```

Program: Checking Planet Names

- The `planet.c` program illustrates how to access command-line arguments.
- The program is designed to check a series of strings to see which ones are names of planets.
- The strings are put on the command line:

```
planet Jupiter venus Earth fred
```

- The program will indicate whether each string is a planet name and, if it is, display the planet's number:

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

planet.c

```
/* Checks planet names */

#include <stdio.h>
#include <string.h>

#define NUM_PLANETS 9

int main(int argc, char *argv[])
{
  char *planets[] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
  int i, j;
```

```
  for (i = 1; i < argc; i++) {
    for (j = 0; j < NUM_PLANETS; j++)
      if (strcmp(argv[i], planets[j]) == 0) {
        printf("%s is planet %d\n", argv[i], j + 1);
        break;
      }
    if (j == NUM_PLANETS)
      printf("%s is not a planet\n", argv[i]);
  }

  return 0;
}
```

Chapter 14

The Preprocessor

Introduction

- Directives such as `#define` and `#include` are handled by the ***preprocessor,*** a piece of software that edits C programs just prior to compilation.
- Its reliance on a preprocessor makes C (along with C++) unique among major programming languages.
- The preprocessor is a powerful tool, but it also can be a source of hard-to-find bugs.

How the Preprocessor Works

- The preprocessor looks for ***preprocessing directives,*** which begin with a `#` character.
- We've encountered the `#define` and `#include` directives before.
- `#define` defines a ***macro***—a name that represents something else, such as a constant.
- The preprocessor responds to a `#define` directive by storing the name of the macro along with its definition.
- When the macro is used later, the preprocessor "expands" the macro, replacing it by its defined value.

How the Preprocessor Works

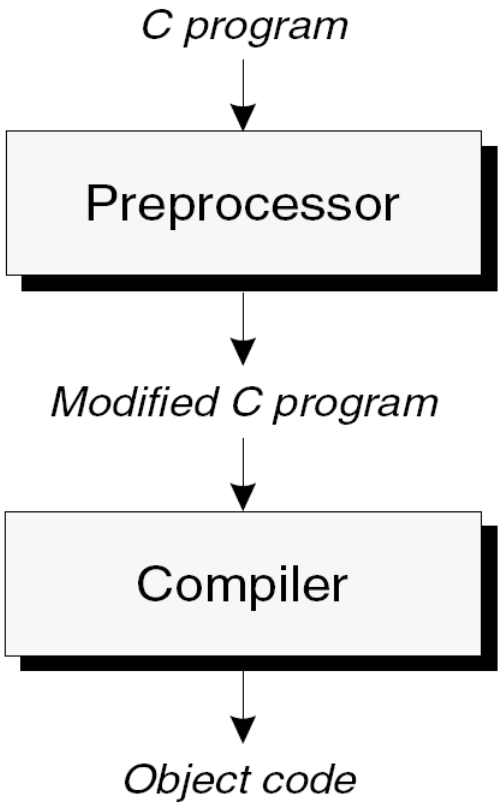
- `#include` tells the preprocessor to open a particular file and "include" its contents as part of the file being compiled.
- For example, the line

  ```
  #include <stdio.h>
  ```

  instructs the preprocessor to open the file named `stdio.h` and bring its contents into the program.

How the Preprocessor Works

- The preprocessor's role in the compilation process:



How the Preprocessor Works

- The input to the preprocessor is a C program, possibly containing directives.
- The preprocessor executes these directives, removing them in the process.
- The preprocessor's output goes directly into the compiler.

How the Preprocessor Works

- The `celsius.c` program of Chapter 2:

```
/* Converts a Fahrenheit temperature to Celsius */

#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
  printf("Celsius equivalent is: %.1f\n", celsius);

  return 0;
}
```

How the Preprocessor Works

- The program after preprocessing:

Blank line
Blank line
Lines brought in from stdio.h
Blank line
Blank line
Blank line
Blank line

```
int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);

  printf("Celsius equivalent is: %.1f\n", celsius);

  return 0;
}
```

How the Preprocessor Works

- The preprocessor does a bit more than just execute directives.
- In particular, it replaces each comment with a single space character.
- Some preprocessors go further and remove unnecessary white-space characters, including spaces and tabs at the beginning of indented lines.

How the Preprocessor Works

- In the early days of C, the preprocessor was a separate program.
- Nowadays, the preprocessor is often part of the compiler, and some of its output may not necessarily be C code.
- Still, it's useful to think of the preprocessor as separate from the compiler.

How the Preprocessor Works

- Most C compilers provide a way to view the output of the preprocessor.
- Some compilers generate preprocessor output when a certain option is specified (GCC will do so when the `-E` option is used).
- Others come with a separate program that behaves like the integrated preprocessor.

How the Preprocessor Works

- A word of caution: The preprocessor has only a limited knowledge of C.
- As a result, it's quite capable of creating illegal programs as it executes directives.
- In complicated programs, examining the output of the preprocessor may prove useful for locating this kind of error.

Preprocessing Directives

- Most preprocessing directives fall into one of three categories:
  - ***Macro definition.*** The `#define` directive defines a macro; the `#undef` directive removes a macro definition.
  - ***File inclusion.*** The `#include` directive causes the contents of a specified file to be included in a program.
  - ***Conditional compilation.*** The `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` directives allow blocks of text to be either included in or excluded from a program.

Preprocessing Directives

- Several rules apply to all directives.
- ***Directives always begin with the # symbol.***

  The # symbol need not be at the beginning of a line, as long as only white space precedes it.
- ***Any number of spaces and horizontal tab characters may separate the tokens in a directive.***

  Example:

```
#    define    N    100
```

Preprocessing Directives

- ***Directives always end at the first new-line character, unless explicitly continued.***

  To continue a directive to the next line, end the current line with a \ character:

```
#define DISK_CAPACITY (SIDES *                  \
                       TRACKS_PER_SIDE *        \
                       SECTORS_PER_TRACK *      \
                       BYTES_PER_SECTOR)
```

Preprocessing Directives

- ***Directives can appear anywhere in a program.***

  Although `#define` and `#include` directives usually appear at the beginning of a file, other directives are more likely to show up later.

- ***Comments may appear on the same line as a directive.***

  It's good practice to put a comment at the end of a macro definition:

```
#define FREEZING_PT 32.0f /* freezing point of water */
```

Macro Definitions

- The macros that we've been using since Chapter 2 are known as *simple* macros, because they have no parameters.
- The preprocessor also supports *parameterized* macros.

Simple Macros

- Definition of a ***simple macro*** (or ***object-like macro***):

  `#define` *identifier replacement-list*

  replacement-list is any sequence of ***preprocessing tokens.***
- The replacement list may include identifiers, keywords, numeric constants, character constants, string literals, operators, and punctuation.
- Wherever *identifier* appears later in the file, the preprocessor substitutes *replacement-list*.

Simple Macros

- Any extra symbols in a macro definition will become part of the replacement list.
- Putting the = symbol in a macro definition is a common error:

```
#define N = 100   /*** WRONG ***/
…
int a[N];         /* becomes int a[= 100]; */
```

Simple Macros

- Ending a macro definition with a semicolon is another popular mistake:

```
#define N 100;  /*** WRONG ***/
…
int a[N];       /* becomes int a[100;]; */
```

- The compiler will detect most errors caused by extra symbols in a macro definition.
- Unfortunately, the compiler will flag each use of the macro as incorrect, rather than identifying the actual culprit: the macro's definition.

Simple Macros

- Simple macros are primarily used for defining "manifest constants"—names that represent numeric, character, and string values:

```
#define STR_LEN 80
#define TRUE    1
#define FALSE   0
#define PI      3.14159
#define CR      '\r'
#define EOS     '\0'
#define MEM_ERR "Error: not enough memory"
```

Simple Macros

- Advantages of using `#define` to create names for constants:
  - ***It makes programs easier to read.*** The name of the macro can help the reader understand the meaning of the constant.
  - ***It makes programs easier to modify.*** We can change the value of a constant throughout a program by modifying a single macro definition.
  - ***It helps avoid inconsistencies and typographical errors.*** If a numerical constant like 3.14159 appears many times in a program, chances are it will occasionally be written 3.1416 or 3.14195 by accident.

Simple Macros

- Simple macros have additional uses.
- ***Making minor changes to the syntax of C***

  Macros can serve as alternate names for C symbols:

```
#define BEGIN {
#define END   }
#define LOOP for (;;)
```

  Changing the syntax of C usually isn't a good idea, since it can make programs harder for others to understand.

Simple Macros

- ***Renaming types***

  An example from Chapter 5:

  ```
  #define BOOL int
  ```

  Type definitions are a better alternative.

- ***Controlling conditional compilation***

  Macros play an important role in controlling conditional compilation.

  A macro that might indicate "debugging mode":

  ```
  #define DEBUG
  ```

Simple Macros

- When macros are used as constants, C programmers customarily capitalize all letters in their names.
- However, there's no consensus as to how to capitalize macros used for other purposes.
  - Some programmers like to draw attention to macros by using all upper-case letters in their names.
  - Others prefer lower-case names, following the style of K&R.

Parameterized Macros

- Definition of a ***parameterized macro*** (also known as a ***function-like macro***):

  `#define` *identifier* `(` x_1 `,` x_2 `,` … `,` x_n `)` *replacement-list*

  x_1, x_2, …, x_n are identifiers (the macro's ***parameters***).
- The parameters may appear as many times as desired in the replacement list.
- There must be *no space* between the macro name and the left parenthesis.
- If space is left, the preprocessor will treat `(`x_1, x_2, …, x_n`)` as part of the replacement list.

Parameterized Macros

- When the preprocessor encounters the definition of a parameterized macro, it stores the definition away for later use.
- Wherever a macro ***invocation*** of the form *identifier* $(y_1, y_2, \ldots, y_n)$ appears later in the program, the preprocessor replaces it with *replacement-list*, substituting y_1 for x_1, y_2 for x_2, and so forth.
- Parameterized macros often serve as simple functions.

Parameterized Macros

- Examples of parameterized macros:

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
#define IS_EVEN(n)  ((n)%2==0)
```

- Invocations of these macros:

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

- The same lines after macro replacement:

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

Parameterized Macros

- A more complicated function-like macro:

```
#define TOUPPER(c) \
  ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

- The `<ctype.h>` header provides a similar function named `toupper` that's more portable.
- A parameterized macro may have an empty parameter list:

```
#define getchar() getc(stdin)
```

- The empty parameter list isn't really needed, but it makes `getchar` resemble a function.

Parameterized Macros

- Using a parameterized macro instead of a true function has a couple of advantages:
  - ***The program may be slightly faster.*** A function call usually requires some overhead during program execution, but a macro invocation does not.
  - ***Macros are "generic."*** A macro can accept arguments of any type, provided that the resulting program is valid.

Parameterized Macros

- Parameterized macros also have disadvantages.
- ***The compiled code will often be larger.***

  Each macro invocation increases the size of the source program (and hence the compiled code).

  The problem is compounded when macro invocations are nested:

```
n = MAX(i, MAX(j, k));
```

  The statement after preprocessing:

```
n = ((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```

Parameterized Macros

- ***Arguments aren't type-checked.***

  When a function is called, the compiler checks each argument to see if it has the appropriate type.

  Macro arguments aren't checked by the preprocessor, nor are they converted.

- ***It's not possible to have a pointer to a macro.***

  C allows pointers to functions, a useful concept.

  Macros are removed during preprocessing, so there's no corresponding notion of "pointer to a macro."

Parameterized Macros

- ***A macro may evaluate its arguments more than once.***

  Unexpected behavior may occur if an argument has side effects:

```
n = MAX(i++, j);
```

  The same line after preprocessing:

```
n = ((i++)>(j)?(i++):(j));
```

  If `i` is larger than `j`, then `i` will be (incorrectly) incremented twice and `n` will be assigned an unexpected value.

Parameterized Macros

- Errors caused by evaluating a macro argument more than once can be difficult to find, because a macro invocation looks the same as a function call.
- To make matters worse, a macro may work properly most of the time, failing only for certain arguments that have side effects.
- For self-protection, it's a good idea to avoid side effects in arguments.

Parameterized Macros

- Parameterized macros can be used as patterns for segments of code that are often repeated.
- A macro that makes it easier to display integers:

```
#define PRINT_INT(n) printf("%d\n", n)
```

- The preprocessor will turn the line

```
PRINT_INT(i/j);
```

into

```
printf("%d\n", i/j);
```

The # Operator

- Macro definitions may contain two special operators, # and ##.
- Neither operator is recognized by the compiler; instead, they're executed during preprocessing.
- The # operator converts a macro argument into a string literal; it can appear only in the replacement list of a parameterized macro.
- The operation performed by # is known as "stringization."

The # Operator

- There are a number of uses for #; let's consider just one.
- Suppose that we decide to use the `PRINT_INT` macro during debugging as a convenient way to print the values of integer variables and expressions.
- The # operator makes it possible for `PRINT_INT` to label each value that it prints.

The # Operator

- Our new version of `PRINT_INT`:

```
#define PRINT_INT(n) printf(#n " = %d\n", n)
```

- The invocation

```
PRINT_INT(i/j);
```

  will become

```
printf("i/j" " = %d\n", i/j);
```

- The compiler automatically joins adjacent string literals, so this statement is equivalent to

```
printf("i/j = %d\n", i/j);
```

The ## Operator

- The ## operator can "paste" two tokens together to form a single token.
- If one of the operands is a macro parameter, pasting occurs after the parameter has been replaced by the corresponding argument.

The ## Operator

- A macro that uses the `##` operator:

```
#define MK_ID(n) i##n
```

- A declaration that invokes `MK_ID` three times:

```
int MK_ID(1), MK_ID(2), MK_ID(3);
```

- The declaration after preprocessing:

```
int i1, i2, i3;
```

The ## Operator

- The `##` operator has a variety of uses.
- Consider the problem of defining a `max` function that behaves like the `MAX` macro described earlier.
- A single `max` function usually isn't enough, because it will only work for arguments of one type.
- Instead, we can write a macro that expands into the definition of a `max` function.
- The macro's parameter will specify the type of the arguments and the return value.

The ## Operator

- There's just one snag: if we use the macro to create more than one function named `max`, the program won't compile.
- To solve this problem, we'll use the `##` operator to create a different name for each version of `max`:

```
#define GENERIC_MAX(type)        \
type type##_max(type x, type y)  \
{                                \
  return x > y ? x : y;          \
}
```

- An invocation of this macro:

```
GENERIC_MAX(float)
```

- The resulting function definition:

```
float float_max(float x, float y) { return x > y ? x : y; }
```

General Properties of Macros

- Several rules apply to both simple and parameterized macros.
- ***A macro's replacement list may contain invocations of other macros.***

  Example:

```
#define PI      3.14159
#define TWO_PI (2*PI)
```

  When it encounters `TWO_PI` later in the program, the preprocessor replaces it by `(2*PI)`.

  The preprocessor then ***rescans*** the replacement list to see if it contains invocations of other macros.

General Properties of Macros

- ***The preprocessor replaces only entire tokens.***

  Macro names embedded in identifiers, character constants, and string literals are ignored.

  Example:

```
#define SIZE 256

int BUFFER_SIZE;

if (BUFFER_SIZE > SIZE)
  puts("Error: SIZE exceeded");
```

  Appearance after preprocessing:

```
int BUFFER_SIZE;

if (BUFFER_SIZE > 256)
  puts("Error: SIZE exceeded");
```

General Properties of Macros

- ***A macro definition normally remains in effect until the end of the file in which it appears.***

  Macros don't obey normal scope rules.

  A macro defined inside the body of a function isn't local to that function; it remains defined until the end of the file.

- ***A macro may not be defined twice unless the new definition is identical to the old one.***

  Differences in spacing are allowed, but the tokens in the macro's replacement list (and the parameters, if any) must be the same.

General Properties of Macros

- ***Macros may be "undefined" by the* `#undef` *directive.***

  The `#undef` directive has the form

  `#undef` *identifier*

  where *identifier* is a macro name.

  One use of `#undef` is to remove the existing definition of a macro so that it can be given a new definition.

Parentheses in Macro Definitions

- The replacement lists in macro definitions often require parentheses in order to avoid unexpected results.
- If the macro's replacement list contains an operator, always enclose the replacement list in parentheses:

```
#define TWO_PI (2*3.14159)
```

- Also, put parentheses around each parameter every time it appears in the replacement list:

```
#define SCALE(x) ((x)*10)
```

- Without the parentheses, we can't guarantee that the compiler will treat replacement lists and arguments as whole expressions.

Parentheses in Macro Definitions

- An example that illustrates the need to put parentheses around a macro's replacement list:

```
#define TWO_PI 2*3.14159
/* needs parentheses around replacement list */
```

- During preprocessing, the statement

```
conversion_factor = 360/TWO_PI;
```

  becomes

```
conversion_factor = 360/2*3.14159;
```

  The division will be performed before the multiplication.

Parentheses in Macro Definitions

- Each occurrence of a parameter in a macro's replacement list needs parentheses as well:

```
#define SCALE(x) (x*10)
  /* needs parentheses around x */
```

- During preprocessing, the statement

```
j = SCALE(i+1);
```

becomes

```
j = (i+1*10);
```

This statement is equivalent to

```
j = i+10;
```

Creating Longer Macros

- The comma operator can be useful for creating more sophisticated macros by allowing us to make the replacement list a series of expressions.
- A macro that reads a string and then prints it:

```
#define ECHO(s) (gets(s), puts(s))
```

- Calls of `gets` and `puts` are expressions, so it's perfectly legal to combine them using the comma operator.
- We can invoke `ECHO` as though it were a function:

```
ECHO(str);  /* becomes (gets(str), puts(str)); */
```

Creating Longer Macros

- An alternative definition of `ECHO` that uses braces:

```
#define ECHO(s) { gets(s); puts(s); }
```

- Suppose that we use `ECHO` in an `if` statement:

```
if (echo_flag)
  ECHO(str);
else
  gets(str);
```

- Replacing `ECHO` gives the following result:

```
if (echo_flag)
  { gets(str); puts(str); };
else
  gets(str);
```

Creating Longer Macros

- The compiler treats the first two lines as a complete `if` statement:

```
if (echo_flag)
  { gets(str); puts(str); }
```

- It treats the semicolon that follows as a null statement and produces an error message for the `else` clause, since it doesn't belong to any `if`.
- We could solve the problem by remembering not to put a semicolon after each invocation of `ECHO`, but then the program would look odd.

Creating Longer Macros

- The comma operator solves this problem for `ECHO`, but not for all macros.
- If a macro needs to contain a series of *statements*, not just a series of *expressions*, the comma operator is of no help.
- The solution is to wrap the statements in a `do` loop whose condition is false:

```
do { … } while (0)
```

- Notice that the `do` statement needs a semicolon at the end.

Creating Longer Macros

- A modified version of the `ECHO` macro:

```
#define ECHO(s)      \
        do {         \
          gets(s);   \
          puts(s);   \
        } while (0)
```

- When `ECHO` is used, it must be followed by a semicolon, which completes the `do` statement:

```
ECHO(str);
/* becomes
   do { gets(str); puts(str); } while (0); */
```

Predefined Macros

- C has several predefined macros, each of which represents an integer constant or string literal.
- The `__DATE__` and `__TIME__` macros identify when a program was compiled.
- Example of using `__DATE__` and `__TIME__`:

```
printf("Wacky Windows (c) 2010 Wacky Software, Inc.\n");
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```

- Output produced by these statements:

```
Wacky Windows (c) 2010 Wacky Software, Inc.
Compiled on Dec 23 2010 at 22:18:48
```

- This information can be helpful for distinguishing among different versions of the same program.

Predefined Macros

- We can use the `__LINE__` and `__FILE__` macros to help locate errors.
- A macro that can help pinpoint the location of a division by zero:

```
#define CHECK_ZERO(divisor) \
  if (divisor == 0) \
    printf("*** Attempt to divide by zero on line %d " \
           "of file %s ***\n", __LINE__, __FILE__)
```

- The `CHECK_ZERO` macro would be invoked prior to a division:

```
CHECK_ZERO(j);
k = i / j;
```

Predefined Macros

- If `j` happens to be zero, a message of the following form will be printed:

```
*** Attempt to divide by zero on line 9 of file foo.c ***
```

- Error-detecting macros like this one are quite useful.
- In fact, the C library has a general-purpose error-detecting macro named `assert`.
- The remaining predefined macro is named `__STDC__`.
- This macro exists and has the value 1 if the compiler conforms to the C standard (either C89 or C99).

Additional Predefined Macros in C99

- C99 provides a few additional predefined macros.
- The `__STDC__HOSTED__` macro represents the constant 1 if the compiler is a hosted implementation. Otherwise, the macro has the value 0.
- An ***implementation*** of C consists of the compiler plus other software necessary to execute C programs.
- A ***hosted implementation*** must accept any program that conforms to the C99 standard.
- A ***freestanding implementation*** doesn't have to compile programs that use complex types or standard headers beyond a few of the most basic.

Additional Predefined Macros in C99

- The `__STDC__VERSION__` macro provides a way to check which version of the C standard is recognized by the compiler.
  - If a compiler conforms to the C89 standard, including Amendment 1, the value is `199409L`.
  - If a compiler conforms to the C99 standard, the value is `199901L`.

Additional Predefined Macros in C99

- A C99 compiler will define up to three additional macros, but only if the compiler meets certain requirements:

  `__STDC_IEC_559__` is defined (and has the value 1) if the compiler performs floating-point arithmetic according to IEC 60559.

  `__STDC_IEC_559_COMPLEX__` is defined (and has the value 1) if the compiler performs complex arithmetic according to IEC 60559.

  `__STDC_ISO_10646__` is defined as *yyyymm*`L` if wide characters are represented by the codes in ISO/IEC 10646 (with revisions as of the specified year and month).

Empty Macro Arguments (C99)

- C99 allows any or all of the arguments in a macro call to be empty.
- Such a call will contain the same number of commas as a normal call.
- Wherever the corresponding parameter name appears in the replacement list, it's replaced by nothing.

Empty Macro Arguments (C99)

- Example:

```
#define ADD(x,y) (x+y)
```

- After preprocessing, the statement

```
i = ADD(j,k);
```

becomes

```
i = (j+k);
```

whereas the statement

```
i = ADD(,k);
```

becomes

```
i = (+k);
```

Empty Macro Arguments (C99)

- When an empty argument is an operand of the # or ## operators, special rules apply.
- If an empty argument is "stringized" by the # operator, the result is "" (the empty string):

```
#define MK_STR(x) #x
…
char empty_string[] = MK_STR();
```

- The declaration after preprocessing:

```
char empty_string[] = "";
```

Empty Macro Arguments (C99)

- If one of the arguments of the ## operator is empty, it's replaced by an invisible "placemarker" token.
- Concatenating an ordinary token with a placemarker token yields the original token (the placemarker disappears).
- If two placemarker tokens are concatenated, the result is a single placemarker.
- Once macro expansion has been completed, placemarker tokens disappear from the program.

Empty Macro Arguments (C99)

- Example:

```
#define JOIN(x,y,z) x##y##z
…
int JOIN(a,b,c), JOIN(a,b,), JOIN(a,,c), JOIN(,,c);
```

- The declaration after preprocessing:

```
int abc, ab, ac, c;
```

- The missing arguments were replaced by placemarker tokens, which then disappeared when concatenated with any nonempty arguments.
- All three arguments to the `JOIN` macro could even be missing, which would yield an empty result.

Macros with a Variable Number of Arguments (C99)

- C99 allows macros that take an unlimited number of arguments.
- A macro of this kind can pass its arguments to a function that accepts a variable number of arguments.
- Example:

```
#define TEST(condition, ...) ((condition)? \
  printf("Passed test: %s\n", #condition): \
  printf(__VA_ARGS__))
```

- The `...` token (***ellipsis***) goes at the end of the parameter list, preceded by ordinary parameters, if any.
- `__VA_ARGS__` is a special identifier that represents all the arguments that correspond to the ellipsis.

Macros with a Variable Number of Arguments (C99)

- An example that uses the `TEST` macro:

```
TEST(voltage <= max_voltage,
     "Voltage %d exceeds %d\n", voltage, max_voltage);
```

- Preprocessor output (reformatted for readability):

```
((voltage <= max_voltage)?
  printf("Passed test: %s\n", "voltage <= max_voltage"):
  printf("Voltage %d exceeds %d\n", voltage, max_voltage));
```

- The program will display the message

```
Passed test: voltage <= max_voltage
```

  if `voltage` is no more than `max_voltage`.

- Otherwise, it will display the values of `voltage` and `max_voltage`:

```
Voltage 125 exceeds 120
```

The `__func__` Identifier (C99)

- The `__func__` identifier behaves like a string variable that stores the name of the currently executing function.
- The effect is the same as if each function contains the following declaration at the beginning of its body:

  `static const char __func__[] = "`*function-name*`";`

  where *function-name* is the name of the function.

The `__func__` Identifier (C99)

- Debugging macros that rely on the `__func__` identifier:

```
#define FUNCTION_CALLED() printf("%s called\n", __func__);
#define FUNCTION_RETURNS() printf("%s returns\n", __func__);
```

- These macros can used to trace function calls:

```
void f(void)
{
  FUNCTION_CALLED();   /* displays "f called" */
  …
  FUNCTION_RETURNS();  /* displays "f returns" */
}
```

- Another use of `__func__`: it can be passed to a function to let it know the name of the function that called it.

Conditional Compilation

- The C preprocessor recognizes a number of directives that support ***conditional compilation.***
- This feature permits the inclusion or exclusion of a section of program text depending on the outcome of a test performed by the preprocessor.

The `#if` and `#endif` Directives

- Suppose we're in the process of debugging a program.
- We'd like the program to print the values of certain variables, so we put calls of `printf` in critical parts of the program.
- Once we've located the bugs, it's often a good idea to let the `printf` calls remain, just in case we need them later.
- Conditional compilation allows us to leave the calls in place, but have the compiler ignore them.

The `#if` and `#endif` Directives

- The first step is to define a macro and give it a nonzero value:

```
#define DEBUG 1
```

- Next, we'll surround each group of `printf` calls by an `#if`-`#endif` pair:

```
#if DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```

The `#if` and `#endif` Directives

- During preprocessing, the `#if` directive will test the value of `DEBUG`.
- Since its value isn't zero, the preprocessor will leave the two calls of `printf` in the program.
- If we change the value of `DEBUG` to zero and recompile the program, the preprocessor will remove all four lines from the program.
- The `#if`-`#endif` blocks can be left in the final program, allowing diagnostic information to be produced later if any problems turn up.

The `#if` and `#endif` Directives

- General form of the `#if` and `#endif` directives:

  `#if` *constant-expression*
  `#endif`
- When the preprocessor encounters the `#if` directive, it evaluates the constant expression.
- If the value of the expression is zero, the lines between `#if` and `#endif` will be removed from the program during preprocessing.
- Otherwise, the lines between `#if` and `#endif` will remain.

The `#if` and `#endif` Directives

- The `#if` directive treats undefined identifiers as macros that have the value 0.
- If we neglect to define `DEBUG`, the test

  ```
  #if DEBUG
  ```

  will fail (but not generate an error message).
- The test

  ```
  #if !DEBUG
  ```

  will succeed.

The `defined` Operator

- The preprocessor supports three operators: `#`, `##`, and `defined`.
- When applied to an identifier, `defined` produces the value 1 if the identifier is a currently defined macro; it produces 0 otherwise.
- The `defined` operator is normally used in conjunction with the `#if` directive.

The `defined` Operator

- Example:

```
#if defined(DEBUG)
…
#endif
```

- The lines between `#if` and `#endif` will be included only if `DEBUG` is defined as a macro.
- The parentheses around `DEBUG` aren't required:

```
#if defined DEBUG
```

- It's not necessary to give `DEBUG` a value:

```
#define DEBUG
```

The `#ifdef` and `#ifndef` Directives

- The `#ifdef` directive tests whether an identifier is currently defined as a macro:

  `#ifdef` *identifier*
- The effect is the same as

  `#if defined(`*identifier*`)`
- The `#ifndef` directive tests whether an identifier is *not* currently defined as a macro:

  `#ifndef` *identifier*
- The effect is the same as

  `#if !defined(`*identifier*`)`

The `#elif` and `#else` Directives

- `#if`, `#ifdef`, and `#ifndef` blocks can be nested just like ordinary `if` statements.
- When nesting occurs, it's a good idea to use an increasing amount of indentation as the level of nesting grows.
- Some programmers put a comment on each closing `#endif` to indicate what condition the matching `#if` tests:

```
#if DEBUG
…
#endif /* DEBUG */
```

The `#elif` and `#else` Directives

- `#elif` and `#else` can be used in conjunction with `#if`, `#ifdef`, or `#ifndef` to test a series of conditions:

  `#if` *expr1*
  Lines to be included if expr1 is nonzero
  `#elif` *expr2*
  Lines to be included if expr1 is zero but expr2 is nonzero
  `#else`
  Lines to be included otherwise
  `#endif`

- Any number of `#elif` directives—but at most one `#else`—may appear between `#if` and `#endif`.

Uses of Conditional Compilation

- Conditional compilation has other uses besides debugging.
- ***Writing programs that are portable to several machines or operating systems.***

  Example:

```
#if defined(WIN32)
…
#elif defined(MAC_OS)
…
#elif defined(LINUX)
…
#endif
```

Uses of Conditional Compilation

- ***Writing programs that can be compiled with different compilers.***

  An example that uses the `__STDC__` macro:

```
#if __STDC__
Function prototypes
#else
Old-style function declarations
#endif
```

  If the compiler does not conform to the C standard, old-style function declarations are used instead of function prototypes.

Uses of Conditional Compilation

- ***Providing a default definition for a macro.***

  Conditional compilation makes it possible to check whether a macro is currently defined and, if not, give it a default definition:

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

Uses of Conditional Compilation

- ***Temporarily disabling code that contains comments.***

  A `/*…*/` comment can't be used to "comment out" code that already contains `/*…*/` comments.

  An `#if` directive can be used instead:

```
#if 0
Lines containing comments
#endif
```

Uses of Conditional Compilation

- Chapter 15 discusses another common use of conditional compilation: protecting header files against multiple inclusion.

Miscellaneous Directives

- The `#error`, `#line`, and `#pragma` directives are more specialized than the ones we've already examined.
- These directives are used much less frequently.

The `#error` Directive

- Form of the `#error` directive:

  `#error` *message*

  message is any sequence of tokens.
- If the preprocessor encounters an `#error` directive, it prints an error message which must include *message*.
- If an `#error` directive is processed, some compilers immediately terminate compilation without attempting to find other errors.

The `#error` Directive

- `#error` directives are frequently used in conjunction with conditional compilation.
- Example that uses an `#error` directive to test the maximum value of the `int` type:

```
#if INT_MAX < 100000
#error int type is too small
#endif
```

The `#error` Directive

- The `#error` directive is often found in the `#else` part of an `#if`-`#elif`-`#else` series:

```
#if defined(WIN32)
…
#elif defined(MAC_OS)
…
#elif defined(LINUX)
…
#else
#error No operating system specified
#endif
```

The `#line` Directive

- The `#line` directive is used to alter the way program lines are numbered.
- First form of the `#line` directive:

  `#line` *n*

  Subsequent lines in the program will be numbered *n*, *n* + 1, *n* + 2, and so forth.
- Second form of the `#line` directive:

  `#line` *n* "*file*"

  Subsequent lines are assumed to come from *file*, with line numbers starting at *n*.

The `#line` Directive

- The `#line` directive changes the value of the `__LINE__` macro (and possibly `__FILE__`).
- Most compilers will use the information from the `#line` directive when generating error messages.
- Suppose that the following directive appears at the beginning of `foo.c`:

  ```
  #line 10 "bar.c"
  ```

  If the compiler detects an error on line 5 of `foo.c`, the message will refer to line 13 of file `bar.c`.
- The `#line` directive is used primarily by programs that generate C code as output.

The `#line` Directive

- The most famous example is `yacc` (Yet Another Compiler-Compiler), a UNIX utility that automatically generates part of a compiler.
- The programmer prepares a file that contains information for `yacc` as well as fragments of C code.
- From this file, `yacc` generates a C program, `y.tab.c`, that incorporates the code supplied by the programmer.
- By inserting `#line` directives, `yacc` tricks the compiler into believing that the code comes from the original file.
- Error messages produced during the compilation of `y.tab.c` will refer to lines in the original file.

The `#pragma` Directive

- The `#pragma` directive provides a way to request special behavior from the compiler.
- Form of a `#pragma` directive:

  `#pragma` *tokens*
- `#pragma` directives can be very simple (a single token) or they can be much more elaborate:

```
#pragma data(heap_size => 1000, stack_size => 2000)
```

The `#pragma` Directive

- The set of commands that can appear in `#pragma` directives is different for each compiler.
- The preprocessor must ignore any `#pragma` directive that contains an unrecognized command; it's not permitted to give an error message.
- In C89, there are no standard pragmas—they're all implementation-defined.
- C99 has three standard pragmas, all of which use `STDC` as the first token following `#pragma`.

The `_Pragma` Operator (C99)

- C99 introduces the `_Pragma` operator, which is used in conjunction with the `#pragma` directive.
- A `_Pragma` expression has the form

  `_Pragma (` *string-literal* `)`
- When it encounters such an expression, the preprocessor "destringizes" the string literal:
  - Double quotes around the string are removed.
  - `\"` is replaced by `"`.
  - `\\` is replaced by `\`.

The `_Pragma` Operator (C99)

- The resulting tokens are then treated as though they appear in a `#pragma` directive.
- For example, writing

```
_Pragma("data(heap_size => 1000, stack_size => 2000)")
```

  is the same as writing

```
#pragma data(heap_size => 1000, stack_size => 2000)
```

The `_Pragma` Operator (C99)

- The `_Pragma` operator lets us work around the fact that a preprocessing directive can't generate another directive.
- `_Pragma`, however, is an operator, not a directive, and can therefore appear in a macro definition.
- This makes it possible for a macro expansion to leave behind a `#pragma` directive.

The `_Pragma` Operator (C99)

- A macro that uses the `_Pragma` operator:

```
#define DO_PRAGMA(x) _Pragma(#x)
```

- An invocation of the macro:

```
DO_PRAGMA(GCC dependency "parse.y")
```

- The result after expansion:

```
#pragma GCC dependency "parse.y"
```

- The tokens passed to `DO_PRAGMA` are stringized into `"GCC dependency \"parse.y\""`.
- The `_Pragma` operator destringizes this string, producing a `#pragma` directive.

Chapter 16

Structures, Unions, and Enumerations

Structure Variables

- The properties of a ***structure*** are different from those of an array.
  - The elements of a structure (its ***members***) aren't required to have the same type.
  - The members of a structure have names; to select a particular member, we specify its name, not its position.
- In some languages, structures are called ***records,*** and members are known as ***fields.***

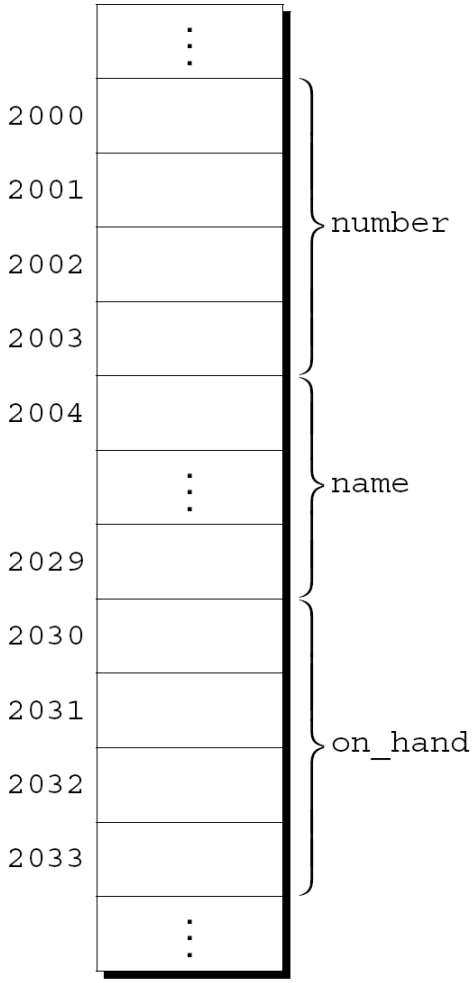
Declaring Structure Variables

- A structure is a logical choice for storing a collection of related data items.
- A declaration of two structure variables that store information about parts in a warehouse:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

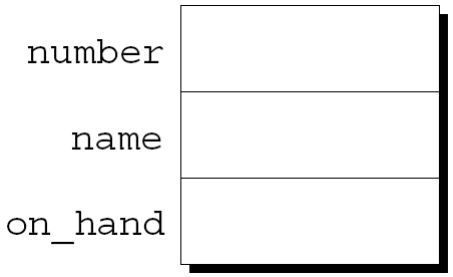
Declaring Structure Variables

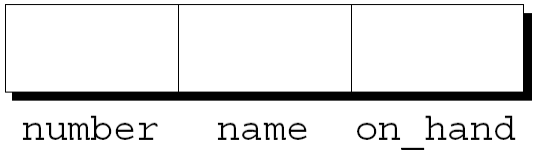
- The members of a structure are stored in memory in the order in which they're declared.
- Appearance of `part1` ⟶
- Assumptions:
  - `part1` is located at address 2000.
  - Integers occupy four bytes.
  - `NAME_LEN` has the value 25.
  - There are no gaps between the members.



Declaring Structure Variables

- Abstract representations of a structure:





- Member values will go in the boxes later.

Declaring Structure Variables

- Each structure represents a new scope.
- Any names declared in that scope won't conflict with other names in a program.
- In C terminology, each structure has a separate ***name space*** for its members.

Declaring Structure Variables

- For example, the following declarations can appear in the same program:

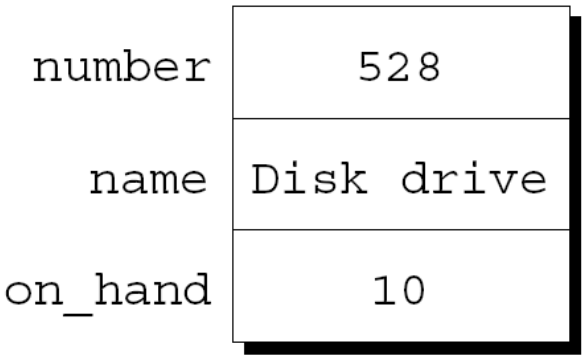
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

struct {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;
```

Initializing Structure Variables

- A structure declaration may include an initializer:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```

- Appearance of `part1` after initialization:



Initializing Structure Variables

- Structure initializers follow rules similar to those for array initializers.
- Expressions used in a structure initializer must be constant. (This restriction is relaxed in C99.)
- An initializer can have fewer members than the structure it's initializing.
- Any "leftover" members are given 0 as their initial value.

Designated Initializers (C99)

- C99's designated initializers can be used with structures.
- The initializer for `part1` shown in the previous example:

```
{528, "Disk drive", 10}
```

- In a designated initializer, each value would be labeled by the name of the member that it initializes:

```
{.number = 528, .name = "Disk drive", .on_hand = 10}
```

- The combination of the period and the member name is called a ***designator.***

Designated Initializers (C99)

- Designated initializers are easier to read and check for correctness.
- Also, values in a designated initializer don't have to be placed in the same order that the members are listed in the structure.
  - The programmer doesn't have to remember the order in which the members were originally declared.
  - The order of the members can be changed in the future without affecting designated initializers.

Designated Initializers (C99)

- Not all values listed in a designated initializer need be prefixed by a designator.
- Example:

```
{.number = 528, "Disk drive", .on_hand = 10}
```

  The compiler assumes that `"Disk drive"` initializes the member that follows `number` in the structure.
- Any members that the initializer fails to account for are set to zero.

Operations on Structures

- To access a member within a structure, we write the name of the structure first, then a period, then the name of the member.
- Statements that display the values of `part1`'s members:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

Operations on Structures

- The members of a structure are lvalues.
- They can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258;
  /* changes part1's part number */
part1.on_hand++;
  /* increments part1's quantity on hand */
```

Operations on Structures

- The period used to access a structure member is actually a C operator.
- It takes precedence over nearly all other operators.
- Example:

```
scanf("%d", &part1.on_hand);
```

The `.` operator takes precedence over the `&` operator, so `&` computes the address of `part1.on_hand`.

Operations on Structures

- The other major structure operation is assignment:

```
part2 = part1;
```

- The effect of this statement is to copy `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.

Operations on Structures

- Arrays can't be copied using the = operator, but an array embedded within a structure is copied when the enclosing structure is copied.
- Some programmers exploit this property by creating "dummy" structures to enclose arrays that will be copied later:

```
struct { int a[10]; } a1, a2;
a1 = a2;
  /* legal, since a1 and a2 are structures */
```

Operations on Structures

- The = operator can be used only with structures of ***compatible*** types.
- Two structures declared at the same time (as `part1` and `part2` were) are compatible.
- Structures declared using the same "structure tag" or the same type name are also compatible.
- Other than assignment, C provides no operations on entire structures.
- In particular, the == and != operators can't be used with structures.

Structure Types

- Suppose that a program needs to declare several structure variables with identical members.
- We need a name that represents a *type* of structure, not a particular structure *variable*.
- Ways to name a structure:
  - Declare a "structure tag"
  - Use `typedef` to define a type name

Declaring a Structure Tag

- A ***structure tag*** is a name used to identify a particular kind of structure.
- The declaration of a structure tag named `part`:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

- Note that a semicolon must follow the right brace.

Declaring a Structure Tag

- The `part` tag can be used to declare variables:

```
struct part part1, part2;
```

- We can't drop the word `struct`:

```
part part1, part2;   /*** WRONG ***/
```

`part` isn't a type name; without the word `struct`, it is meaningless.

- Since structure tags aren't recognized unless preceded by the word `struct`, they don't conflict with other names used in a program.

Declaring a Structure Tag

- The declaration of a structure *tag* can be combined with the declaration of structure *variables*:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Declaring a Structure Tag

- All structures declared to have type `struct part` are compatible with one another:

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;

part2 = part1;
  /* legal; both parts have the same type */
```

Defining a Structure Type

- As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name.
- A definition of a type named `Part`:

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

- `Part` can be used in the same way as the built-in types:

```
Part part1, part2;
```

Defining a Structure Type

- When it comes time to name a structure, we can usually choose either to declare a structure tag or to use `typedef`.
- However, declaring a structure tag is mandatory when the structure is to be used in a linked list (Chapter 17).

Structures as Arguments and Return Values

- Functions may have structures as arguments and return values.
- A function with a structure argument:

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

- A call of `print_part`:

```
print_part(part1);
```

Structures as Arguments and Return Values

- A function that returns a `part` structure:

```
struct part build_part(int number,
                       const char *name,
                       int on_hand)
{
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

- A call of `build_part`:

```
part1 = build_part(528, "Disk drive", 10);
```

Structures as Arguments and Return Values

- Passing a structure to a function and returning a structure from a function both require making a copy of all members in the structure.
- To avoid this overhead, it's sometimes advisable to pass a pointer to a structure or return a pointer to a structure.
- Chapter 17 gives examples of functions that have a pointer to a structure as an argument and/or return a pointer to a structure.

Structures as Arguments and Return Values

- There are other reasons to avoid copying structures.
- For example, the `<stdio.h>` header defines a type named `FILE`, which is typically a structure.
- Each `FILE` structure stores information about the state of an open file and therefore must be unique in a program.
- Every function in `<stdio.h>` that opens a file returns a pointer to a `FILE` structure.
- Every function that performs an operation on an open file requires a `FILE` pointer as an argument.

Structures as Arguments and Return Values

- Within a function, the initializer for a structure variable can be another structure:

```
void f(struct part part1)
{
  struct part part2 = part1;
  …
}
```

- The structure being initialized must have automatic storage duration.

Compound Literals (C99)

- Chapter 9 introduced the C99 feature known as the ***compound literal.***
- A compound literal can be used to create a structure "on the fly," without first storing it in a variable.
- The resulting structure can be passed as a parameter, returned by a function, or assigned to a variable.

Compound Literals (C99)

- A compound literal can be used to create a structure that will be passed to a function:

```
print_part((struct part) {528, "Disk drive", 10});
```

  The compound literal is shown in **bold**.
- A compound literal can also be assigned to a variable:

```
part1 = (struct part) {528, "Disk drive", 10};
```

- A compound literal consists of a type name within parentheses, followed by a set of values in braces.
- When a compound literal represents a structure, the type name can be a structure tag preceded by the word `struct` or a `typedef` name.

Compound Literals (C99)

- A compound literal may contain designators, just like a designated initializer:

```
print_part((struct part) {.on_hand = 10,
                          .name = "Disk drive",
                          .number = 528});
```

- A compound literal may fail to provide full initialization, in which case any uninitialized members default to zero.

Nested Arrays and Structures

- Structures and arrays can be combined without restriction.
- Arrays may have structures as their elements, and structures may contain arrays and structures as members.

Nested Structures

- Nesting one structure inside another is often useful.
- Suppose that `person_name` is the following structure:

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```

Nested Structures

- We can use `person_name` as part of a larger structure:

```
struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

- Accessing `student1`'s first name, middle initial, or last name requires two applications of the . operator:

```
strcpy(student1.name.first, "Fred");
```

Nested Structures

- Having `name` be a structure makes it easier to treat names as units of data.
- A function that displays a name could be passed one `person_name` argument instead of three arguments:

```
display_name(student1.name);
```

- Copying the information from a `person_name` structure to the `name` member of a student structure would take one assignment instead of three:

```
struct person_name new_name;
…
student1.name = new_name;
```

Arrays of Structures

- One of the most common combinations of arrays and structures is an array whose elements are structures.
- This kind of array can serve as a simple database.
- An array of `part` structures capable of storing information about 100 parts:

```
struct part inventory[100];
```

Arrays of Structures

- Accessing a part in the array is done by using subscripting:

```
print_part(inventory[i]);
```

- Accessing a member within a `part` structure requires a combination of subscripting and member selection:

```
inventory[i].number = 883;
```

- Accessing a single character in a part name requires subscripting, followed by selection, followed by subscripting:

```
inventory[i].name[0] = '\0';
```

Initializing an Array of Structures

- Initializing an array of structures is done in much the same way as initializing a multidimensional array.
- Each structure has its own brace-enclosed initializer; the array initializer wraps another set of braces around the structure initializers.

Initializing an Array of Structures

- One reason for initializing an array of structures is that it contains information that won't change during program execution.
- Example: an array that contains country codes used when making international telephone calls.
- The elements of the array will be structures that store the name of a country along with its code:

```
struct dialing_code {
  char *country;
  int code;
};
```

Initializing an Array of Structures

```
const struct dialing_code country_codes[] =
  {{"Argentina",             54}, {"Bangladesh",       880},
   {"Brazil",                55}, {"Burma (Myanmar)",   95},
   {"China",                 86}, {"Colombia",          57},
   {"Congo, Dem. Rep. of", 243}, {"Egypt",             20},
   {"Ethiopia",            251}, {"France",            33},
   {"Germany",              49}, {"India",             91},
   {"Indonesia",            62}, {"Iran",              98},
   {"Italy",                39}, {"Japan",             81},
   {"Mexico",               52}, {"Nigeria",          234},
   {"Pakistan",             92}, {"Philippines",       63},
   {"Poland",               48}, {"Russia",             7},
   {"South Africa",         27}, {"South Korea",       82},
   {"Spain",                34}, {"Sudan",            249},
   {"Thailand",             66}, {"Turkey",            90},
   {"Ukraine",             380}, {"United Kingdom",    44},
   {"United States",         1}, {"Vietnam",           84}};
```

- The inner braces around each structure value are optional.

Initializing an Array of Structures

- C99's designated initializers allow an item to have more than one designator.
- A declaration of the `inventory` array that uses a designated initializer to create a single part:

```
struct part inventory[100] =
  {[0].number = 528, [0].on_hand = 10,
   [0].name[0] = '\0'};
```

The first two items in the initializer use two designators; the last item uses three.

Program: Maintaining a Parts Database

- The `inventory.c` program illustrates how nested arrays and structures are used in practice.
- The program tracks parts stored in a warehouse.
- Information about the parts is stored in an array of structures.
- Contents of each structure:
  - Part number
  - Name
  - Quantity

Program: Maintaining a Parts Database

- Operations supported by the program:
  - Add a new part number, part name, and initial quantity on hand
  - Given a part number, print the name of the part and the current quantity on hand
  - Given a part number, change the quantity on hand
  - Print a table showing all information in the database
  - Terminate program execution

Program: Maintaining a Parts Database

- The codes `i` (insert), `s` (search), `u` (update), `p` (print), and `q` (quit) will be used to represent these operations.
- A session with the program:

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10
```

Program: Maintaining a Parts Database

```
Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2
```

Program: Maintaining a Parts Database

```
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8

Enter operation code: p
Part Number   Part Name                 Quantity on Hand
    528       Disk drive                       8
    914       Printer cable                    5

Enter operation code: q
```

Program: Maintaining a Parts Database

- The program will store information about each part in a structure.
- The structures will be stored in an array named `inventory`.
- A variable named `num_parts` will keep track of the number of parts currently stored in the array.

Program: Maintaining a Parts Database

- An outline of the program's main loop:

```
for (;;) {
   prompt user to enter operation code;
   read code;
   switch (code) {
     case 'i': perform insert operation; break;
     case 's': perform search operation; break;
     case 'u': perform update operation; break;
     case 'p': perform print operation; break;
     case 'q': terminate program;
     default:  print error message;
   }
}
```

Program: Maintaining a Parts Database

- Separate functions will perform the insert, search, update, and print operations.
- Since the functions will all need access to `inventory` and `num_parts`, these variables will be external.
- The program is split into three files:
  - `inventory.c` (the bulk of the program)
  - `readline.h` (contains the prototype for the `read_line` function)
  - `readline.c` (contains the definition of `read_line`)

inventory.c

```
/* Maintains a parts database (array version) */

#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0;   /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```

```
/**********************************************************
 * main: Prompts the user to enter an operation code,     *
 *       then calls a function to perform the requested   *
 *       action. Repeats until the user enters the        *
 *       command 'q'. Prints an error message if the user *
 *       enters an illegal code.                          *
 **********************************************************/
int main(void)
{
  char code;
  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line */
      ;
```

```
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}
```

```
/**********************************************************
 * find_part: Looks up a part number in the inventory     *
 *            array. Returns the array index if the part  *
 *            number is found; otherwise, returns -1.     *
 **********************************************************/
int find_part(int number)
{
  int i;

  for (i = 0; i < num_parts; i++)
    if (inventory[i].number == number)
      return i;
  return -1;
}
```

```
/**********************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         database. Prints an error message and returns  *
 *         prematurely if the part already exists or the  *
 *         database is full.                              *
 **********************************************************/
void insert(void)
{
  int part_number;

  if (num_parts == MAX_PARTS) {
    printf("Database is full; can't add more parts.\n");
    return;
  }
```

```
  printf("Enter part number: ");
  scanf("%d", &part_number);
  if (find_part(part_number) >= 0) {
    printf("Part already exists.\n");
    return;
  }

  inventory[num_parts].number = part_number;
  printf("Enter part name: ");
  read_line(inventory[num_parts].name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &inventory[num_parts].on_hand);
  num_parts++;
}
```

```
/**********************************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 **********************************************************/
void search(void)
{
  int i, number;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Part name: %s\n", inventory[i].name);
    printf("Quantity on hand: %d\n", inventory[i].on_hand);
  } else
    printf("Part not found.\n");
}
```

```
/**********************************************************
 * update: Prompts the user to enter a part number.       *
 *         Prints an error message if the part doesn't    *
 *         exist; otherwise, prompts the user to enter    *
 *         change in quantity on hand and updates the     *
 *         database.                                      *
 **********************************************************/
void update(void)
{
  int i, number, change;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    inventory[i].on_hand += change;
  } else
    printf("Part not found.\n");
}
```

```
/**********************************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Parts are printed in the      *
 *        order in which they were entered into the       *
 *        database.                                       *
 **********************************************************/
void print(void)
{
  int i;

  printf("Part Number   Part Name                  "
         "Quantity on Hand\n");
  for (i = 0; i < num_parts; i++)
    printf("%7d       %-25s%11d\n", inventory[i].number,
           inventory[i].name, inventory[i].on_hand);
}
```

Program: Maintaining a Parts Database

- The version of `read_line` in Chapter 13 won't work properly in the current program.
- Consider what happens when the user inserts a part:

```
Enter part number: 528
Enter part name: Disk drive
```

- The user presses the Enter key after entering the part number, leaving an invisible new-line character that the program must read.
- When `scanf` reads the part number, it consumes the 5, 2, and 8, but leaves the new-line character unread.

Program: Maintaining a Parts Database

- If we try to read the part name using the original `read_line` function, it will encounter the new-line character immediately and stop reading.
- This problem is common when numerical input is followed by character input.
- One solution is to write a version of `read_line` that skips white-space characters before it begins storing characters.
- This solves the new-line problem and also allows us to avoid storing blanks that precede the part name.

readline.h

```
#ifndef READLINE_H
#define READLINE_H

/**********************************************************
 * read_line: Skips leading white-space characters, then  *
 *            reads the remainder of the input line and   *
 *            stores it in str. Truncates the line if its *
 *            length exceeds n. Returns the number of     *
 *            characters stored.                          *
 **********************************************************/
int read_line(char str[], int n);

#endif
```

readline.c

```
#include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
  int ch, i = 0;

  while (isspace(ch = getchar()))
    ;
  while (ch != '\n' && ch != EOF) {
    if (i < n)
      str[i++] = ch;
    ch = getchar();
  }
  str[i] = '\0';
  return i;
}
```

Unions

- A ***union,*** like a structure, consists of one or more members, possibly of different types.
- The compiler allocates only enough space for the largest of the members, which overlay each other within this space.
- Assigning a new value to one member alters the values of the other members as well.

Unions

- An example of a union variable:

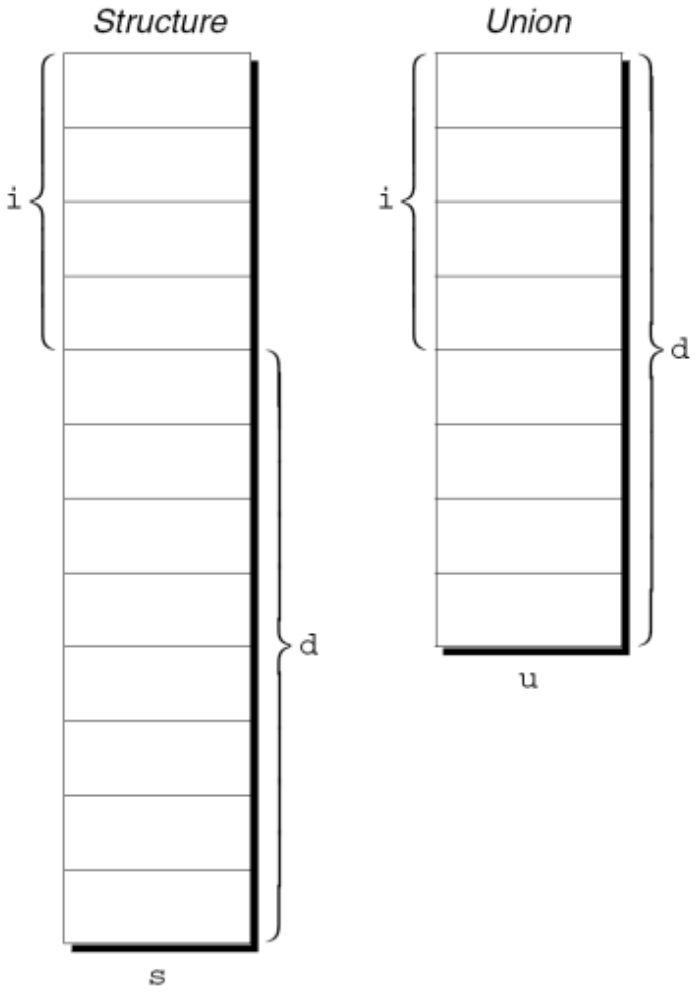
```
union {
  int i;
  double d;
} u;
```

- The declaration of a union closely resembles a structure declaration:

```
struct {
  int i;
  double d;
} s;
```

Unions

- The structure `s` and the union `u` differ in just one way.
- The members of `s` are stored at different addresses in memory.
- The members of `u` are stored at the same address.



Unions

- Members of a union are accessed in the same way as members of a structure:

```
u.i = 82;
u.d = 74.8;
```

- Changing one member of a union alters any value previously stored in any of the other members.
  - Storing a value in `u.d` causes any value previously stored in `u.i` to be lost.
  - Changing `u.i` corrupts `u.d`.

Unions

- The properties of unions are almost identical to the properties of structures.
- We can declare union tags and union types in the same way we declare structure tags and types.
- Like structures, unions can be copied using the = operator, passed to functions, and returned by functions.

Unions

- Only the first member of a union can be given an initial value.
- How to initialize the `i` member of `u` to 0:

```
union {
  int i;
  double d;
} u = {0};
```

- The expression inside the braces must be constant. (The rules are slightly different in C99.)

Unions

- Designated initializers can also be used with unions.
- A designated initializer allows us to specify which member of a union should be initialized:

```
union {
  int i;
  double d;
} u = {.d = 10.0};
```

- Only one member can be initialized, but it doesn't have to be the first one.

Unions

- Applications for unions:
  - Saving space
  - Building mixed data structures
  - Viewing storage in different ways (discussed in Chapter 20)

Using Unions to Save Space

- Unions can be used to save space in structures.
- Suppose that we're designing a structure that will contain information about an item that's sold through a gift catalog.
- Each item has a stock number and a price, as well as other information that depends on the type of the item:

  Books: Title, author, number of pages

  Mugs: Design

  Shirts: Design, colors available, sizes available

Using Unions to Save Space

- A first attempt at designing the `catalog_item` structure:

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  char title[TITLE_LEN+1];
  char author[AUTHOR_LEN+1];
  int num_pages;
  char design[DESIGN_LEN+1];
  int colors;
  int sizes;
};
```

Using Unions to Save Space

- The `item_type` member would have one of the values `BOOK`, `MUG`, or `SHIRT`.
- The `colors` and `sizes` members would store encoded combinations of colors and sizes.
- This structure wastes space, since only part of the information in the structure is common to all items in the catalog.
- By putting a union inside the `catalog_item` structure, we can reduce the space required by the structure.

Using Unions to Save Space

```
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  union {
    struct {
      char title[TITLE_LEN+1];
      char author[AUTHOR_LEN+1];
      int num_pages;
    } book;
    struct {
      char design[DESIGN_LEN+1];
    } mug;
    struct {
      char design[DESIGN_LEN+1];
      int colors;
      int sizes;
    } shirt;
  } item;
};
```

Using Unions to Save Space

- If `c` is a `catalog_item` structure that represents a book, we can print the book's title in the following way:

```
printf("%s", c.item.book.title);
```

- As this example shows, accessing a union that's nested inside a structure can be awkward.

Using Unions to Save Space

- The `catalog_item` structure can be used to illustrate an interesting aspect of unions.
- Normally, it's not a good idea to store a value into one member of a union and then access the data through a different member.
- However, there is a special case: two or more of the members of the union are structures, and the structures begin with one or more matching members.
- If one of the structures is currently valid, then the matching members in the other structures will also be valid.

Using Unions to Save Space

- The union embedded in the `catalog_item` structure contains three structures as members.
- Two of these (`mug` and `shirt`) begin with a matching member (`design`).
- Now, suppose that we assign a value to one of the `design` members:

```
strcpy(c.item.mug.design, "Cats");
```

- The `design` member in the other structure will be defined and have the same value:

```
printf("%s", c.item.shirt.design);
  /* prints "Cats" */
```

Using Unions to Build Mixed Data Structures

- Unions can be used to create data structures that contain a mixture of data of different types.
- Suppose that we need an array whose elements are a mixture of `int` and `double` values.
- First, we define a union type whose members represent the different kinds of data to be stored in the array:

```
typedef union {
  int i;
  double d;
} Number;
```

Using Unions to Build Mixed Data Structures

- Next, we create an array whose elements are `Number` values:

```
Number number_array[1000];
```

- A `Number` union can store either an `int` value or a `double` value.
- This makes it possible to store a mixture of `int` and `double` values in `number_array`:

```
number_array[0].i = 5;
number_array[1].d = 8.395;
```

Adding a "Tag Field" to a Union

- There's no easy way to tell which member of a union was last changed and therefore contains a meaningful value.
- Consider the problem of writing a function that displays the value stored in a `Number` union:

```
void print_number(Number n)
{
  if (n contains an integer)
    printf("%d", n.i);
  else
    printf("%g", n.d);
}
```

There's no way for `print_number` to determine whether `n` contains an integer or a floating-point number.

Adding a "Tag Field" to a Union

- In order to keep track of this information, we can embed the union within a structure that has one other member: a "tag field" or "discriminant."
- The purpose of a tag field is to remind us what's currently stored in the union.
- `item_type` served this purpose in the `catalog_item` structure.

Adding a "Tag Field" to a Union

- The `Number` type as a structure with an embedded union:

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct {
  int kind;   /* tag field */
  union {
    int i;
    double d;
  } u;
} Number;
```

- The value of `kind` will be either `INT_KIND` or `DOUBLE_KIND`.

Adding a "Tag Field" to a Union

- Each time we assign a value to a member of `u`, we'll also change `kind` to remind us which member of `u` we modified.
- An example that assigns a value to the `i` member of `u`:

```
n.kind = INT_KIND;
n.u.i = 82;
```

`n` is assumed to be a `Number` variable.

Adding a "Tag Field" to a Union

- When the number stored in a `Number` variable is retrieved, `kind` will tell us which member of the union was the last to be assigned a value.
- A function that takes advantage of this capability:

```
void print_number(Number n)
{
  if (n.kind == INT_KIND)
    printf("%d", n.u.i);
  else
    printf("%g", n.u.d);
}
```

Enumerations

- In many programs, we'll need variables that have only a small set of meaningful values.
- A variable that stores the suit of a playing card should have only four potential values: "clubs," "diamonds," "hearts," and "spades."

Enumerations

- A "suit" variable can be declared as an integer, with a set of codes that represent the possible values of the variable:

```
int s;    /* s will store a suit */
…
s = 2;    /* 2 represents "hearts" */
```

- Problems with this technique:
  - We can't tell that `s` has only four possible values.
  - The significance of 2 isn't apparent.

Enumerations

- Using macros to define a suit "type" and names for the various suits is a step in the right direction:

```
#define SUIT     int
#define CLUBS    0
#define DIAMONDS 1
#define HEARTS   2
#define SPADES   3
```

- An updated version of the previous example:

```
SUIT s;
…
s = HEARTS;
```

Enumerations

- Problems with this technique:
  - There's no indication to someone reading the program that the macros represent values of the same "type."
  - If the number of possible values is more than a few, defining a separate macro for each will be tedious.
  - The names `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` will be removed by the preprocessor, so they won't be available during debugging.

Enumerations

- C provides a special kind of type designed specifically for variables that have a small number of possible values.
- An ***enumerated type*** is a type whose values are listed ("enumerated") by the programmer.
- Each value must have a name (an ***enumeration constant***).

Enumerations

- Although enumerations have little in common with structures and unions, they're declared in a similar way:

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

- The names of enumeration constants must be different from other identifiers declared in the enclosing scope.

Enumerations

- Enumeration constants are similar to constants created with the `#define` directive, but they're not equivalent.
- If an enumeration is declared inside a function, its constants won't be visible outside the function.

Enumeration Tags and Type Names

- As with structures and unions, there are two ways to name an enumeration: by declaring a tag or by using `typedef` to create a genuine type name.
- Enumeration tags resemble structure and union tags:

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

- `suit` variables would be declared in the following way:

```
enum suit s1, s2;
```

Enumeration Tags and Type Names

- As an alternative, we could use `typedef` to make `Suit` a type name:

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
Suit s1, s2;
```

- In C89, using `typedef` to name an enumeration is an excellent way to create a Boolean type:

```
typedef enum {FALSE, TRUE} Bool;
```

Enumerations as Integers

- Behind the scenes, C treats enumeration variables and constants as integers.
- By default, the compiler assigns the integers 0, 1, 2, … to the constants in a particular enumeration.
- In the `suit` enumeration, `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` represent 0, 1, 2, and 3, respectively.

Enumerations as Integers

- The programmer can choose different values for enumeration constants:

```
enum suit {CLUBS = 1, DIAMONDS = 2,
           HEARTS = 3, SPADES = 4};
```

- The values of enumeration constants may be arbitrary integers, listed in no particular order:

```
enum dept {RESEARCH = 20,
           PRODUCTION = 10, SALES = 25};
```

- It's even legal for two or more enumeration constants to have the same value.

Enumerations as Integers

- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant.
- The first enumeration constant has the value 0 by default.
- Example:

```
enum EGA_colors {BLACK, LT_GRAY = 7,
                 DK_GRAY, WHITE = 15};
```

`BLACK` has the value 0, `LT_GRAY` is 7, `DK_GRAY` is 8, and `WHITE` is 15.

Enumerations as Integers

- Enumeration values can be mixed with ordinary integers:

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;

i = DIAMONDS;    /* i is now 1                */
s = 0;           /* s is now 0 (CLUBS)        */
s++;             /* s is now 1 (DIAMONDS)     */
i = s + 2;       /* i is now 3                */
```

- `s` is treated as a variable of some integer type.
- `CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES` are names for the integers 0, 1, 2, and 3.

Enumerations as Integers

- Although it's convenient to be able to use an enumeration value as an integer, it's dangerous to use an integer as an enumeration value.
- For example, we might accidentally store the number 4—which doesn't correspond to any suit—into `s`.

Using Enumerations to Declare "Tag Fields"

- Enumerations are perfect for determining which member of a union was the last to be assigned a value.
- In the `Number` structure, we can make the `kind` member an enumeration instead of an `int`:

```
typedef struct {
  enum {INT_KIND, DOUBLE_KIND} kind;
  union {
    int i;
    double d;
  } u;
} Number;
```

Using Enumerations to Declare "Tag Fields"

- The new structure is used in exactly the same way as the old one.
- Advantages of the new structure:
  - Does away with the `INT_KIND` and `DOUBLE_KIND` macros
  - Makes it obvious that `kind` has only two possible values: `INT_KIND` and `DOUBLE_KIND`